版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

异步图书
www.epubit.com.cn

Pearson

Docker Containers Build and Deploy with Kubernetes, Flannel, Cockpit, and Atomic

Docker容器

利用Kubernetes、Flannel、Cockpit和Atomic构建和部署

[美] Christopher Negus 著　　任发科 王东 译



中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

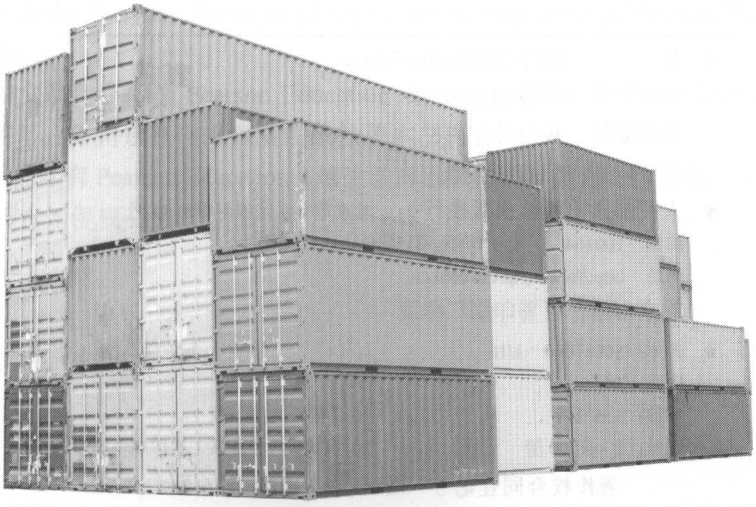
Pearson

Docker Containers Build and Deploy with Kubernetes, Flannel, Cockpit, and Atomic

Docker容器

利用Kubernetes、Flannel、Cockpit和Atomic构建和部署

[美] Christopher Negus 著　　任发科 王东 译



人民邮电出版社

北京

图书在版编目（CIP）数据

Docker容器 : 利用Kubernetes、Flannel、Cockpit和Atomic构建和部署 / (美) 克里斯托弗·尼格斯(Christopher Negus) 著 ; 任发科, 王东译. -- 北京 : 人民邮电出版社, 2017.5
书名原文: Docker Containers: Build and Deploy with Kubernetes, Flannel, Cockpit, and Atomic
ISBN 978-7-115-45105-7

Ⅰ. ①D… Ⅱ. ①克… ②任… ③王… Ⅲ. ①Linux操作系统－程序设计 Ⅳ. ①TP316.85

中国版本图书馆CIP数据核字(2017)第052959号

内 容 提 要

本书是 Linux 系统及云环境上运行 Docker 的实用指南。书中全面讲解创建、运行、停止、启动、保存和管理容器的具体方法，同时也涉及了多容器管理的主题。本书内容分成 5 个部分。第一部分开启容器之旅，介绍开始使用 Docker 容器所需了解的知识；第二部分关注单个容器，主要介绍如何通过 docker 命令直接使用单个容器；第三部分主题是在云环境上运行容器，描述如何运行超级特权容器（SPC），以及如何使用 Cockpit 在云环境或者本地环境下跨多个宿主机管理容器；第四部分重点是管理多容器，探究容器的编排（利用 Kubernates）；第五部分专注于开发容器，描述一些开发 Docker 容器的建议和技巧，最后通过展示一些 Dockerfile 文件阐述如何构建容器。

本书是写给系统管理员、软件开发人员、运维人员和技术狂热者的，书中触及 Docker 使用的方方面面。阅读本书不需要读者了解 Docker 或者容器化方面的知识，读者可以将本书作为 Docker 的入门书。当然，本书也探讨了更高级的 Docker 话题，如编排和容器的开发。

◆ 著　[美] Christopher Negus
译　任发科　王 东
责任编辑　杨海玲
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京市昌平百善印刷厂印刷

◆ 开本：800×1000　1/16
印张：14.75
字数：315 千字　2017 年 5 月第 1 版
印数：1－3 000 册　2017 年 5 月北京第 1 次印刷
著作权合同登记号　图字：01-2016-3946 号

定价：59.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广字第 8052 号

一如既往，谨以此书献给我的爱妻 Sheree。

——Christopher Negus

译者序

自互联网诞生以来，其简单、快速的产品开发理念和实践渐成主流，并迅速突破互联网行业向软件行业的各个方面波及开来。然而，这种表面的高歌猛进的背后却是龙鱼混杂、泥沙俱下。一方面，有些人将手拉肩扛的“996”当成文化，追求通过简单粗暴的方式维持较高的执行力；另一方面，有些人致力于寻找更为优秀的流程及工具，希冀通过它们来高效地应对越来越复杂的问题。无涉对错，追求进步的努力从未稍作停歇。

回溯软件开发短短几十年的历史，自第一次软件危机以来，寻找高效而优雅的开发方法是一个一路陪伴软件行业成长的问题。从早期的软件工程，到以 CMMI 为代表的过程改进，再到以精益为宗的敏捷系列方法，在面向交付的领域，我们已经摸索出一套研发对接业务的有效方法。而后互联网的兴起，交付不再是软件开发生命周期的终点，部署和运维成为软件开发所需关注的新阶段。正是在此前提下，DevOps 应时而起，其意在给出一个将代码变更高效地推送到生产环境的方式。但有别于过程优化，技术在部署和运维领域始终是最为关键的因素。而这其中，对软件运行环境的抽象和管理一直是整个部署和运维高效化的难点。

在 Docker 之前，环境管理与应用管理通常被分割开来，交给不同团队负责并通过变更流程进行协调。虽然 Ansible、Puppet 这类配置管理工具能够将环境信息抽象出来并以描述性的方式进行配置，从而大大降低环境管理的难度。但从持续部署的角度看，部署流水线需要将环境部署和应用部署统一起来，以便实现一键式的高效自动化部署。那些早期互联网企业，已在其成长过程中逐步构建起自己的整个部署生态。例如，亚马逊通过 Brazil Build 和 Apollo 系统可以完成每日 25 000 次以上的部署工作，而 Google 和 Facebook 也通过各自的工具可以达到每日 5 000 次左右的部署。但对于中小型互联网公司而言，从头自己研发一套这样的系统投入依然巨大。

Docker 这种轻量级容器技术，不但可以将环境与应用打包分发，而且可以通过描述性的方式进行构建，这无疑使部署流水线中最复杂的问题变得简单。加上 Kubernetes 这样的容器集群管理技术，中小型互联网公司得以以较低的成本完成自己的部署工具链，从而在部署和运维方面可以和互联网巨头一样高效。

毋庸多言，现代 IT 企业的发展走到了一个加速分化的时代。将优秀产品推向用户的成本

和速度将成为决定企业生存与否的关键因素之一。优秀的互联网企业已装备精良并持续优化，你的企业是要日以继夜地赶工，还是要周期性地吐旧纳新？

工具已在手中，时不我待，行动起来……

任发科（常新居士）

2017 年 2 月 28 日于柔克学院

译者简介



任发科　网名常新居士，曾任职于亚马逊中国，从事 SDE 和 SDM 的工作。具有十余年的企业软件架构、开发和管理经验，了解软件研发与运营中各环节的工作，目前从事大数据分析工具以及 DevOps 工具链的研发，同时关注 IT 研发管理和团队建设。



王东　毕业于天津大学，毕业后一直从事软件相关研发和架构设计工作，曾在普元软件任资深架构师、在 IBM GBS 任咨询经理、在亚马逊任架构师等，后加入创业公司，从事研发和管理工作；热爱编程，喜欢钻研新技术，对微服务、企业架构、大数据以及 DevOps 有浓厚的兴趣。

前言

Docker 这一容器化技术目前正处于新浪潮的中心，这一浪潮波及了应用的构建、打包和部署。它有可能影响计算机技术的方方面面，从应用程序的开发流程到应用程序如何部署以及跨大规模数据中心进行垂直和水平扩展。

尽管 Docker 非常流行，但它依然是一个非常新的项目，许多人并没有真正理解什么是 Docker。如果你也是其中一员，那么本书会帮你迈出第一步，并让你见识到容器化所承诺的巨大潜力。我的目标是通过本书引领你进入容器化的世界，这些目标可以概括为以下几种方式。

- **通过实践来学习**：我常在自己的书中提及这一点，我相信学习技术最好的方式就是获取并使用它。为了做到这一点，我让你从几种流行的 Linux 系统中进行选择，并介绍如何在所选的系统上安装 Docker，而且还给出使用 Docker 做各种事情的实际例子——从运行简单的容器到构建和管理自己的容器镜像。之后学习会延伸至编排和管理容器的工具和技术。
- **如何通过 Docker 受益**：我描述了在容器中创建和运行应用程序的好处，而不是安装软件包（如 RPM 或者 Deb 格式）并直接从硬盘上运行非容器化的应用程序。除了运行应用程序，我还描述了容器化如何为软件开发人员和系统管理人员带来益处。
- **Docker 的本质特性**：我描述了 Docker 如何利用诸如 Linux 容器（LXC）技术保持容器与宿主机上运行的其他应用程序隔离或者如何有选择性地介入宿主机系统。这些特性包括 Docker 如何使用命名空间、元数据以及独立的文件系统来管理容器化应用并保证其安全。

阅读本书并不需要读者了解 Docker 或者容器化方面的知识，读者可以将本书作为 Docker 的入门书。当然，本书也探讨了更高级的 Docker 话题，如编排和容器的开发。

随着阅读的深入，读者将看到运行、调查、停止和启动、保存以及管理容器的具体方法。开始创建容器时，我讨论了一些技巧，这些技巧将有助于读者创建高效地构建和运行的容器镜像。我还将带读者逐步研究其他人为了生成自己的容器而创建的构建文件（其被称为 Dockerfile）。

阅读本书大体上并不需要 Linux 容器，或者 Docker 容器的知识。不过，话虽如此，有些

技术在 Docker 容器内部和外部都会用到，了解其中的一些会有助于更加得心应手地使用 Docker。

对使用 Docker 有帮助的知识

了解一些 Docker 运行环境的知识有助于更好地使用 Docker 容器。Docker 构建在 Linux 技术之上，并与特定的高级特性相结合，包括用于管理 Linux 命名空间的 Linux 容器（LXC），以及用于管理容器对系统资源（如 CPU 和内存）的访问的 Cgroups。

即使是与 Docker 容器最基本的交互，也依赖于底层的 Linux 技术。你可能听说过 Docker 可以运行在 Windows 或者 Mac 系统之上，但在这些系统上运行 Docker 始终依赖 Linux 虚拟机。换句话说，没有 Linux 就没有 Docker 容器。同样，每个容器自身通常都是从基础镜像构建的，而这个基础镜像又是从特定的 Linux 发行版创建的。

因此，如果没有 Linux 系统的使用经验，你会发现，了解 Linux 的如下方面及相关技术会很有用。

- **命令 shell**：虽然有用来使用 Docker 的图形界面，但本书中的大多数 Docker 的例子都是通过 Linux 命令行 shell 完成的。了解如何使用 Linux shell 可以更高效地使用 Docker。
- **软件包**：Docker 本身是一种软件的交付机制，软件会被打包并作为一个套装应用一同交付。然而，为了构建容器镜像，多数 Docker 基础镜像都允许从其所基于的特定 Linux 发行版安装软件包。

 因此，对于 Ubuntu 基础镜像，需要了解如何通过 apt-get 之类的工具安装 Deb 包。对于 Fedora、Red Hat Enterprise Linux 或者 CentOS 等的 Docker 镜像，yum、dnf 和 rpm 命令非常有用。当你使用这些基础镜像构建自己的 Docker 容器时，这些基础镜像通常能从在线的软件 repository 中自动获取你请求的软件包。了解在你选择的 Linux 发行版上如何获取和安装软件包对于成功使用 Docker 是至关重要的。
- **文件所有权和权限**：Linux 系统中以及容器中的每一个文件都由特定的用户和组所拥有，并设定了某种权限来访问这些文件。有时你想授权在容器中访问来自宿主机的文件和目录。它们中的一些可能是特殊文件，如设备或套接字，这些都是运行应用程序所需要的。进程也需要作为特定的用户运行。了解这些权限如何工作对于容器的正确工作至关重要。

我只提及了有效使用 Docker 容器所需了解的一部分比较显而易见的特性。随着不断探索如何最好地利用自己使用和创建的 Docker 容器，你会接触到很多其他 Linux 相关的特性。

如果不熟悉 Linux，强烈建议你上一门 Linux 基础知识的课程或者读一本讲解 Linux 基础知识的书，以便使用 Docker 容器。我谨建议你阅读本人所著的《Linux Bible》（第 9 版）（http://www.wiley.com/WileyCDA/WileyTitle/productCd-1118999878.html）。这本书不但在构建 Docker 容器所需的技术方面能对你有具体帮助，而且能在开发 Docker 容器镜像时帮你应对

Linux 环境下的工作。

本书涵盖的内容

对于刚开始使用 Docker 容器的人来说，本书要从头至尾地阅读。之后，它可以当作参考资料，提示你与 Docker 容器相关的不同选项和特性。本书内容分成 5 个部分。

第一部分　开启容器之旅

在第一部分中，将学习开始使用 Docker 容器所需了解的知识。第 1 章将描述什么是容器，以及容器与非容器化应用的差别。在第 2 章中，将学习如何在通用 Liunx 系统（如 Fedora 和 Ubuntu）以及面向容器的特定 Linux 系统（如 CoreOS 和 Project Atomic）上安装 Docker。在第 3 章中，我们将展示如何通过配置私有 Docker registry 来保存自己的 Docker 镜像，以此来完成一个基本的容器设置。

第二部分　玩转单个容器

这部分主要涉及通过 docker 命令直接使用单个容器。第 4 章中将展示如何运行你的第一个容器镜像。为了帮你查找并获取容器镜像，第 5 章会描述如何从 Docker registry 搜索容器镜像，然后拉取想要的镜像，将它保存到文件，并将该镜像加载到其他 Docker 系统中。

第 6 章中将学习如何为镜像添加标签，从而更好地识别镜像所包含的内容，并利用这些信息将镜像推送到 registry。第 7 章中将展示如何探查容器或容器镜像的内部，看一下容器或镜像工作方式的细节。第 8 章中将学习如何停止、启动和重启容器。

第 9 章中将学习如何通过将宿主机的目录挂载到容器中来配置存储。为了学习如何配置容器的网络，第 10 章将描述如何配置 Docker 服务通常使用的默认网络（或不使用网络），以及运行容器的人为单个容器配置网络接口的方法。

为了可能的重用，Docker 缓存了大量数据。第 11 章将展示如何清理创建或者运行容器镜像时遗留下来的缓存数据。第 12 章将学习如何构建 Docker 容器，包括高效构建并运行的容器是如何创建的。

第三部分　在云环境上运行容器

第 13 章将描述如何运行所谓的超级特权容器（super privileged container，SPC）。为了阐述超级特权容器如何工作，我会展示怎样获取那些可以在 RHEL Atomic 系统上完成不同管理任务的镜像。第 14 章将描述如何使用 Cockpit（基于 Web 的容器管理工具）在云环境或者本地环境下跨多个宿主机管理容器。

第四部分　管理多容器

在这一部分，我们将探究容器的编排。第 15 章将描述如何在一个系统中使用 Kubernates 的 master 和 node 服务，以便能够尝试 Kubernetes。第 16 章我们将超越一体化 Kubernetes 系统，描述如何搭建 Kubernetes 集群。在 Kubernetes 集群就位后，可以通过 master 计算机将容器 pod 中的应用部署到不同的 node 计算机上进行管理。

第五部分　开发容器

在 Docker 出现的很短的一段时间里，能够更加高效地构建容器的技术就已经被开发出来了。第 17 章将描述一些开发 Docker 容器的建议和技巧。最后，第 18 章通过展示我接触的一些 Dockerfile 文件阐述不同的人是如何克服障碍来构建他们自己的容器的。

如果已经准备好了，马上开始阅读第 1 章吧。希望你喜欢这本书！

致谢

我在创作本书的过程中获得了非常多的帮助。日常工作中，我有幸直接与 Red Hat 的人员一同工作，他们在 Docker、Kubernetes 和 Atomic 这样的项目上完成了卓越的工作，并将这些项目扩展和集成到可以用于最严格的企业环境的操作系统中。总之，我想感谢 Red Hat Enterprise Linux Atomic、OpenShift 以及 Linux 容器团队的开发人员、测试人员和其他作者，感谢他们帮助我每天了解有哪些东西使 Linux 容器适用于企业环境。

我想单独提一下来自 Red Hat 的一些人，他们对本书有着直接的影响。首先，William Henry 写了本书关于存储和网络的这两章。能让他来写这些关键章节，我真是幸运之极。除了这些工作，William 还为 Docker 相关的项目做出了卓越的贡献。实际上，许多 docker 命令的联机帮助页都是 William 写的，这些帮助页随 Docker 软件本身一同交付。再者，有 William 一同参与确定本书的内容，这帮助也是无价的。

本书另一个来自 Red Hat 的重要贡献者是 Scott Collier。Scott 对 Docker 一般知识的公开贡献包括关于 Docker 和 Kubernetes 设置的博客，以及通过 Fedora 云计划分享许多 Dockerfile 示例。就本书而言，Scott 十分慷慨地贡献了他的时间，帮我整理了整本书插图说明的技术和示例。

因为是用业余时间写这本书（这就是花费的时间比我预期要长的原因），所以我常常依赖晚上和周末的时间与出版商（Pearson）进行互动。因此，感谢编辑 Chris Zahn 和 Elaine Wiley 审查本书内容，偶尔要在周日晚上给出回复，并且压缩他们的日程安排以便满足我的日程安排。还有来自 Pearson 的我亲爱的朋友 Debra Williams Cauley，他与我一起开发了这个项目，当我想方设法要将满满当当的日程与我想花时间去写其他书籍进行平衡时，他表现出了极大的耐心。

最后，我想感谢我的家人。当一个人写书时，他无论如何会有一段时间忽视他的家庭。我真的为你们所有人感到自豪。尽管我迷迷糊糊地去写作，我儿子 Seth 却作为校园剧的主演在高中音乐节上出色地模仿了 Zac Efron，他还为自己的高中足球队进行了表演。我儿子 Caleb 找到了自己称心的工作，把家安在了缅因州的自己的小有机农场。而我的妻子 Sheree，继续以健身和斯巴达式运动激励着年轻一代。你们的爱和支持是让我持续前进的动力。

作者介绍

Christopher Negus 是 Linux 的畅销书作者、认证的 Linux 教练和考官、Red Hat 认证架构师和 Red Hat 首席技术作家。目前，Christopher 从事的项目包括 Red Hat OpenStack Platform High Availability、Red Hat Enterprise Linux Atomic Enterprise、Kubernetes 及 Docker 格式的 Linux 容器。

作为一个作者，Christopher 已经写了十多本有关 Linux 和开源软件的书。他的《Linux Bible》（第 9 版，2015 年出版）如今一直位列最畅销的 Linux 书。在互联网爆发的时候，Christopher 的《Red Hat Linux Bible》（第 8 版）卖出超过 25 万册，并两次被选为年度最佳 Linux 图书。Christopher 撰写或合著的其他著作还包括《Linux Toolbox》系列、《Linux Toys》系列、《Fedora and Red Hat Enterprise Linux Bible》系列以及 Wiley 出版的《Linux Troubleshooting Bible》。

对于 Prentice Hall 出版社，Christopher 帮助出版了 Negus 软件解决方案丛书。该丛书中，Christopher 撰写了《Live Linux CDs》并合著了《The Official Damn Small Linux Book》。该丛书还包括关于软件开发、Google Apps 和虚拟化方面的书。

Christopher 在 2008 年作为 RHCE 教练加入 Red Hat。针对这个角色，他成为 Red Hat 认证教练（RHCI）和 Red Hat 认证考官（RHCX）。2014 年，Christopher 成为 Red Hat 认证架构师（RHCA），认证涵盖虚拟化管理、部署和系统管理、集群和存储管理，以及服务器硬化（Server Hardening）。2011 年，Christopher 从 Linux 教练的角色转换回 Red Hat 全职作者，从那时起一直做到现在。

目录

第一部分　开启容器之旅

第 1 章　使用 Docker 对应用进行容器化 …… 2
1.1　了解容器化应用的优缺点 …… 2
1.1.1　容器化应用不是直接在宿主机上运行的应用 …… 3
1.1.2　容器化应用不是直接在虚拟机上运行的应用 …… 3
1.1.3　了解容器的优点 …… 3
1.1.4　了解容器化应用面临的挑战 …… 4
1.2　了解容器的组成 …… 5
1.2.1　Docker 项目 …… 5
1.2.2　Docker Hub Registry …… 6
1.2.3　Docker 镜像和容器 …… 7
1.2.4　docker 命令 …… 7
1.3　探究容器 …… 9
1.4　小结 …… 10
第 2 章　搭建容器运行时环境 …… 11
2.1　标准 Linux 系统的 Docker 配置 …… 12
2.1.1　Ubuntu 的 Docker 配置 …… 12
2.1.2　Fedora 的 Docker 配置 …… 14
2.1.3　Red Hat Enterprise Linux 的 Docker 配置 …… 17
2.1.4　其他操作系统的 Docker 配置 …… 19
2.2　容器式 Linux 系统的 Docker 配置 …… 20
2.2.1　Atomic Host 的 Docker 配置 …… 21
2.2.2　CoreOS 的 Docker 配置 …… 23
2.3　小结 …… 25
第 3 章　搭建私有 Docker Registry …… 26
3.1　获取和启动私有 Docker registry …… 27
3.1.1　在 Fedora 中搭建 Docker registry …… 27
3.1.2　在 Ubuntu 中搭建 Docker registry …… 30
3.2　配置私有 Docker registry …… 32
3.2.1　配置 docker-registry 包 …… 32
3.2.2　配置 registry 容器 …… 35
3.3　了解 Docker 镜像命名空间 …… 35
3.4　小结 …… 37

第二部分　玩转单个容器

第 4 章　运行容器镜像 …… 40
4.1　以交互方式运行容器镜像 …… 42
4.1.1　启动交互 Bash shell …… 42
4.1.2　玩些基于字符的游戏 …… 44

4.1.3 在容器内运行管理命令 45
4.2 运行容器化服务 46
4.2.1 运行容器化 Web 服务器 46
4.2.2 在容器中运行服务时进行资源限制 48
4.3 运行特权容器 49
4.4 小结 50
第 5 章 查找、拉取、保存和装载容器镜像 51
5.1 搜索镜像 51
5.1.1 用 docker 命令搜索镜像 52
5.1.2 在 Docker Hub 上检索镜像 54
5.2 其他 repository 55
5.3 从 Registry 拉取镜像 57
5.4 保存和装载镜像 60
5.5 小结 61
第 6 章 为镜像添加标签 62
6.1 为镜像分配名称 63
6.2 为镜像分配标签 64
6.3 为镜像分配 repository 名 65
6.3.1 为镜像附加用户名 65
6.3.2 为镜像附加 repository 名 67
6.4 小结 68
第 7 章 探究容器 69
7.1 检视镜像和容器 69
7.2 检视镜像 70
7.2.1 用 docker inspect 检视基础镜像 70
7.2.2 用 docker inspect 检视应用程序镜像 72
7.2.3 查看镜像的历史 73
7.3 检视正在运行的容器 73
7.3.1 启动要检视的容器 73
7.3.2 检视整个容器配置 74
7.3.3 检视单个容器属性 78
7.3.4 检视运行终端会话的容器 79
7.3.5 检视容器的内存和 CPU 限制 80
7.3.6 检视容器的 SELinux 上下文 81
7.4 寻找探查容器的更多方法 82
7.4.1 用 docker top 查看容器进程 82
7.4.2 用 docker attach 与容器内服务进行交互 83
7.4.3 用 docker exec 在正在运行的容器中启动新进程 83
7.4.4 用 docker logs 查看容器进程的输出 84
7.4.5 用 docker diff 查看容器的变化 84
7.4.6 用 docker cp 从容器复制文件 85
7.5 小结 85
第 8 章 容器的启动、停止与重启 86
8.1 停止和启动容器 86
8.1.1 停止和启动分离的容器 87
8.1.2 启动和停止交互式容器 88
8.2 重启容器 89
8.3 向容器发送信号 90
8.4 暂停容器与取消暂停 91
8.5 等待容器的退出代码 91
8.6 重命名容器 92
8.7 创建容器 92
8.8 小结 93
第 9 章 配置容器的存储 95
9.1 管理容器的存储 95
9.1.1 使用宿主机的卷 95
9.1.2 数据卷容器 97

9.1.3 对挂载写保护 97
9.1.4 挂载设备 98
9.1.5 挂载套接字 98
9.2 Docker 宿主机的存储策略 99
9.2.1 附加外部存储到 Docker 宿主机 101
9.2.2 使用逻辑卷管理器扩展存储 101
9.3 小结 103
第 10 章 配置容器网络 104
10.1 将端口暴露给其他容器 105
10.2 将端口映射到宿主机外部 106
10.2.1 从被链接的容器映射端口 106
10.2.2 链接不同宿主机上的容器 108
10.3 docker0 网桥的替代方法 109
10.3.1 改变容器的网络模式 109
10.3.2 细查网络选项 109
10.4 修改 Docker 网桥 111
10.5 小结 112
第 11 章 清理容器 113
11.1 为镜像和容器准备空间 113
11.2 删除镜像 114
11.2.1 删除单个镜像 114
11.2.2 删除多个镜像 116
11.3 删除容器 117
11.3.1 删除单个容器 117
11.3.2 删除多个容器 119
11.4 清理和保存容器 119
11.4.1 清理和保存 Ubuntu 容器 120
11.4.2 清理和保存 Fedora 容器 120
11.5 小结 121
第 12 章 构建容器镜像 122
12.1 进行简单的 Docker 构建 123
12.2 通过 Dockerfile 设置要执行的命令 125
12.2.1 使用 CMD 指令 125
12.2.2 使用 ENTRYPOINT 指令 126
12.2.3 使用 RUN 指令 127
12.2.4 通过 Dockerfile 向镜像添加文件 128
12.3 通过 Dockerfile 暴露镜像的端口 128
12.4 在 Dockerfile 中设置环境变量 129
12.5 在 Dockerfile 中设置标签 130
12.6 使用其他 docker build 命令选项 131
12.7 构建容器的技巧 132
12.7.1 清理镜像 132
12.7.2 保持构建目录较小 132
12.7.3 保持容器简单 133
12.7.4 管理进行缓存的方法 133
12.8 小结 133

第三部分 在云环境上运行容器

第 13 章 使用超级特权容器 136
13.1 在 Atomic Host 中使用超级特权容器 137
13.2 了解超级特权容器 137
13.2.1 开放给宿主机的权限 137
13.2.2 访问宿主机的进程表 137
13.2.3 访问宿主机的网络接口 138
13.2.4 访问宿主机的进程间通信 139

13.2.5 访问宿主机的文件系统… 139
13.3 准备使用超级特权容器…… 140
13.4 使用 atomic 命令 …… 140
13.4.1 用 atomic 安装 SPC 镜像…… 141
13.4.2 用 atomic 获取 SPC 镜像的信息…… 141
13.4.3 用 atomic 运行 SPC 镜像…… 142
13.4.4 用 atomic 停止和重启 SPC…… 143
13.4.5 更新 SPC 镜像 …… 143
13.4.6 卸载 SPC 镜像 …… 144
13.5 尝试一些 SPC…… 144
13.5.1 运行 RHEL 工具 SPC … 144
13.5.2 运行日志记录（rsyslog）SPC…… 145
13.5.3 运行系统监控（sadc）SPC …… 147
13.6 小结 …… 149
第 14 章 用 Cockpit 管理云中的容器 … 150
14.1 了解 Cockpit …… 151
14.2 着手使用 Cockpit …… 154
14.3 将服务器添加到 Cockpit …… 155
14.4 从 Cockpit 管理容器 …… 156
14.4.1 向 Cockpit 添加容器镜像 …… 156
14.4.2 从 Cockpit 运行镜像 …… 157
14.5 从 Cockpit 管理网络接口 …… 159
14.6 从 Cockpit 配置存储 …… 161
14.7 在 Cockpit 中做其他管理工作 …… 162
14.8 在 Cockpit 中打开终端 …… 162
14.9 小结 …… 163

第四部分 管理多容器

第 15 章 使用 Kubernetes 进行容器编排 …… 166
15.1 了解 Kubernetes …… 167
15.2 开始使用 Kubernetes …… 168
15.3 搭建一体化 Kubernetes …… 169
15.3.1 安装并启动 Kubernetes… 170
15.3.2 在 Kubernetes 中启动一个 pod…… 171
15.3.3 使用 Kubernetes …… 173
15.4 小结…… 174
第 16 章 创建 Kubernetes 集群…… 176
16.1 了解 Kubernetes 的高级特性 …… 176
16.2 搭建 Kubernetes 集群…… 177
16.2.1 第 1 步：安装 Linux…… 177
16.2.2 第 2 步：搭建 Kubernetes master …… 178
16.2.3 第 3 步：搭建 Kubernetes node …… 180
16.2.4 第 4 步：用 Flannel 进行网络设置 …… 181
16.3 启动 Kubernetes 集群中的 pod … 182
16.4 删除复制控制器、服务和 pod … 186
16.5 小结 …… 187

第五部分 开发容器

第 17 章 开发 Docker 容器…… 190
17.1 为容器开发做好准备…… 190
17.1.1 为 Red Hat 系统选择容器开发环境 …… 190
17.1.2 来自 Docker 的容器开发环境 …… 194

17.2 采用良好的开发实践 …… 195
17.2.1 将文件包含到构建中或排除出去 …… 195
17.2.2 利用分层 …… 196
17.2.3 管理构建中的软件包 …… 197
17.3 进一步了解容器构建 …… 198
17.4 小结 …… 198
第 18 章 探索 Dockerfile 文件示例 …… 199
18.1 细察官方 Docker 镜像的 Dockerfile …… 200
18.1.1 查看 CentOS 的 Dockerfile …… 200
18.1.2 查看 busybox 的 Dockerfile …… 202
18.2 细察来自开源项目的 Dockerfile …… 203
18.2.1 查看 WordPress 的 Dockerfile …… 203
18.2.2 查看 MySQL 的 Dockerfile …… 205
18.3 细察桌面和个人使用的 Dockerfile …… 207
18.3.1 查看 Chrome 的 Dockerfile …… 207
18.3.2 查看 Firefox 的 Dockerfile …… 210
18.4 小结 …… 213

第一部分

开启容器之旅

第 1 章

使用 Docker 对应用进行容器化

本章主要内容

- 了解 Docker 容器
- 了解容器化的挑战
- 了解构成 Docker 的组件
- 开启企业级容器之旅

Docker 为应用程序的打包和运行提供了一种优雅的方式。使用喜欢的 Linux 系统，几分钟之内就能将 Docker 安装好并作为服务运行起来。构建、运行、停止、启动、调查、修改或者用其他的方式操作容器非常容易，说实话，很棒。

Docker 的简单易用使其成为当今最流行的开源项目之一。但是作为数据中心容器化核心的 Docker 却引起了极大的震动，其潜力无异于重新发明了个人和公司（或大或小）创建、测试、部署和管理其最关键应用程序的方式。

使用容器化技术也可以让应用程序向云环境的部署变得更为高效。就像容器本身一样，运行容器的操作系统也能够被瘦身。因为容器已经持有应用程序运行所需的大部分依赖，所以这些用于容器的新型宿主机操作系统就不再需要包含所有依赖了。

本书不仅介绍什么是 Docker 以及 Docker 的工作方式，还揭示了特定用途下扩展 Docker 的若干方法。尽管一个容器的创建和部署是很简单的，但要让多个容器协同工作、访问其他容器的资源，以及访问运行容器的宿主机的资源，则需要更高的复杂性。本书给出了几种创建和部署这些复杂容器集的不同方法。

本书并不只停留在理论层面，其探讨的容器及容器相关工具的各个方面都备有说明这些特性工作方式的实际例子。读者可以自己先尝试一下，然后按照自己使用 Docker 的方式修改和扩展这些实例。

然而，在开始本书的学习之前，读者需要弄清楚关注容器的原因。

1.1 了解容器化应用的优缺点

Docker 提供了一种创建和运行已经在容器中进行了配置的应用程序的方法。想要真正理

解它，先了解容器化应用不是什么是很有帮助的。

1.1.1 容器化应用不是直接在宿主机上运行的应用

运行应用程序的传统方法会将应用程序直接安装到宿主计算机的文件系统上并从那里运行它。从应用程序的视角看来，其环境包含宿主机的进程表、文件系统、IPC 设施、网络接口、端口及设备。

要让应用程序运行起来，通常需要安装与应用程序搭配的额外软件包。一般来说，这不是问题。但有些情况下，可能想在同一个系统上运行相同软件包的不同版本，这就可能会引起冲突。

应用程序与应用程序之间也会以某种方式发生冲突。如果应用程序是服务，它可能会默认绑定特定的网络端口。在服务启动时，它可能还会读取公共配置文件。这会导致无法在同一宿主机上运行该服务的多个实例，或者至少非常棘手；这还让那些想要绑定到同一端口的其他服务难以运行。

直接在宿主机上运行应用程序还有一个缺点，那就是难以迁移应用程序。如果宿主机需要关机或者应用程序需要更多计算能力——超出宿主机所能提供的，那么从宿主计算机上获取所有依赖并将其迁移到另一台宿主机上绝非易事。

1.1.2 容器化应用不是直接在虚拟机上运行的应用

创建虚拟机来运行应用程序能够克服直接在宿主机操作系统上运行应用程序所具有的缺点。虽然虚拟机位于宿主机上，但它作为独立的操作系统运行，它包括自己的内核、文件系统、网络接口等。这样可以很容易将几乎所有东西保存在独立于宿主机的操作系统中。

因为虚拟机是独立的实体，所以不会出现那种直接在硬件上运行应用程序所产生的缺乏灵活性的弊端。可以在宿主机上启动 10 个不同的虚拟机来运行应用程序 10 次。虽然每个虚拟机上的服务监听了同一个端口号，但是因为每个虚拟机拥有不同的 IP 地址，所以并不会引起冲突。

同样地，如果需要关闭宿主计算机，可以将虚拟机迁移到其他宿主机上（如果虚拟化环境支持迁移）或者直接关闭虚拟机并在新宿主机上再次启动它。

一个虚拟机运行一个应用程序的实例的缺点是耗费资源。你的应用程序可能只需要几兆字节磁盘空间来运行，但是整个虚拟机却要耗费许多 GB 的空间。再者，虚拟机的启动时间和 CPU 占用几乎肯定会比应用程序自身消耗的高得多。

容器提供了另一种在宿主机上或虚拟机内直接运行应用程序的方式，这种方式能使应用程序更快、可移植性更好，并且更具可扩展性。

1.1.3 了解容器的优点

就运行应用程序而言，容器有望灵活高效地使用资源。

灵活性来自容器可以包含其所需的全部文件。如同运行于虚拟机中的应用程序，其可以拥有自己的配置文件和依赖库，还可以拥有自己的网络接口——这些网络接口不同于宿主机上配置的那些网络接口。因此，与在虚拟机上运行应用程序一样，容器化应用比直接安装的应用程序更容易迁移，而且因为应用程序运行所栖身的每个容器均拥有独立的网络接口，所以也不会争用同一端口号。

就启动时间、磁盘空间占用和处理能力而言，容器既没有运行独立的操作系统，也没有包含运行整个操作系统所需的大量软件。这是因为容器只包含运行应用程序所需的软件，以及其他想随容器一起运行的工具和少量描述容器的元数据。

与虚拟机不同，Docker 容器并不包含独立的内核（kernel）。Docker 容器中运行的命令会出现在宿主机的进程表中，多数情况下，看起来非常像系统中运行的其他进程。然而，这两种环境中应用程序运行的差异与这两个应用程序所看到的不同世界有着极大的关系。

- **文件系统**：容器拥有自己的文件系统，默认情况下，它无法看到宿主机系统的文件系统。该规则的一个例外是，有些文件（如`/etc/hosts`和`/etc/resolv.conf`）可能会被自动挂载到容器中。另一个例外是，当运行容器镜像时，可以显式地将宿主机的目录挂载到容器中。
- **进程表**：Linux 宿主机上可能运行着成百上千的进程。然而，默认情况下，容器内的进程无法看到宿主机的进程表，容器拥有自己的进程表。因此，你启动容器时所运行的应用程序，其进程在容器内被分配的 PID 为 1。从容器内，进程无法看到宿主机上运行的其他进程——如果这些进程没有在容器内启动的话。
- **网络接口**：默认情况下，Docker 守护进程通过 DHCP 从一组私有 IP 地址集中确定 IP 地址。除了使用 DHCP，Docker 也支持其他网络模式，例如，允许容器使用其他容器的网络接口，直接使用宿主机的网络接口，或者不使用网络接口。依据你的选择，你可以将容器内的端口暴露给宿主机的同一端口或不同端口。
- **IPC 设施**：容器内运行的进程不能与宿主机系统上运行的进程间通信（IPC）设施直接交互。你可以将宿主机上的 IPC 设施暴露给容器，但这不是默认的。每个容器都有自己的 IPC 设施。
- **设备**：容器内的进程无法直接看到宿主机系统上的设备。不过，当容器启动时可以设置一个特殊权限选项来授予该项权限。

如你所见，Docker 容器可以在宿主机的视线内运行，但其运行方式限制了容器越过边界所能看到的宿主机内容（除非明确地开放了那些视图）。

1.1.4 了解容器化应用面临的挑战

容器化应用面临的挑战在于它们不同于那些不在容器中的应用程序。每个 Linux 系统中，用于启动和停止服务以及查看出错消息的设施已就位。Linux 还提供了监控服务和轮换日志文件的方法。

对于运行虚拟机而言，整个虚拟化平台（如 OpenStack 和 Red Hat 企业虚拟化）被构建

来启动、停止和使用虚拟机。虽然管理容器组的工具已在努力构建中，但大多数尚处于萌芽阶段。诸如 Kubernetes 和 OpenShift 这样的项目正在完成部署和管理容器组的框架。

Docker 容器被打包为容器镜像。为了能将容器镜像保存在 registry 中并用 `docker` 命令管理它们，人们已经进行了大量的工作。然而，用于管理 Docker 镜像的工具远未像用来管理 Linux 软件包（如基于 Linux RPM 或 Deb 的系统）的工具那样成熟。

能够验证镜像来自何处的工具、确定镜像是否被篡改的工具以及查看容器中具体安装了哪些软件包和这些软件包的版本的工具才刚刚开发。目前，要意识到，多数情况下很难完全保证从 Docker Hub Registry 上随便获取的镜像可以安全使用。

使用容器的另一个挑战来自这样一个事实：就其本质而言，容器默认情况下无法看到其他容器。因此，要是你想让自己的容器与其他容器紧密配合要怎么办呢？例如，你可能有一个 Web 服务器，你想让这个服务器访问你的数据库服务器。

让容器看到彼此的一些解决方案是 Docker 中能让你将容器连接到一起的特性，以及让你在 pod 的容器之间识别出服务的使用方和提供方的 Kubernetes 的特性。更多容器管理工具正被推出来解决这些问题。但切记，它们尚处于早期开发阶段，而且容器管理的几乎各个领域都有多个工具在开发，有时还相互矛盾。

1.2 了解容器的组成

Docker 是 Docker 项目开发的一种容器格式。`docker` 命令能够运行、停止、启动、调查容器，还能操纵容器。`docker` 命令也可以作为服务守护进程运行，处理管理 Docker 容器的请求。默认情况下，这个 Docker 服务会从 Docker Hub Registry 获取你请求的镜像。虽然你无需知道更多就可以开始使用，但接下来会依次给出一些额外的信息。

1.2.1 Docker 项目

Docker 项目（https://www.docker.com）为 Docker 开发提供了一个中心。它将 Docker 称为“一个针对分布式应用开发者和系统管理员的开放平台”，旨在简化应用程序的开发和分发。

Solomon Hykes 是 Docker 的创始人和 CTO。他将 Docker 要在软件行业做的事情与物理集装箱为航运业所做的事情进行了对比。无论是运输汽车、圆桶、箱子，还是运输钢琴，只要使用标准的集装箱来运输这些不同类型的物品，用来运送和处理它们的工具也可以被标准化。

因此，就其核心而言，Docker 项目提供了一种软件容器的格式并创建了一个专为使用该格式的软件而搭建的简单的基础设施。随着项目的进行，它开始超出其最初所关注的稳定 Docker 格式以及提供工具来管理单个容器。

如今，Docker 项目正将其范围扩展至包含配置与编排工具，来帮助人们部署和管理成组的容器。该项目还研究管理计算资源的方式，并提供高可用性的方式来帮助运行 Docker 容器。随着这些工具变得可用，它们将不得不与谷歌和 Red Hat 这样的公司正开发的更成熟的容器

编排工具（包括本书涵盖的 Kubernetes 项目的工具）正面交锋。

然而，目前 Docker 项目的最大成就是 Docker 容器格式、管理单个容器的工具，以及在 Docker 客户端和 registry 之间拉取和推送 Docker 容器镜像的能力。Docker 项目管理的中央 registry 被称为 Docker Hub Registry。

1.2.2 Docker Hub Registry

Docker Hub Registry（https://registry.hub.docker.com）提供了个人与组织保存和开发其 Docker 容器镜像的地方。当你的 Linux 系统上安装了 Docker 的时候，如果你请求的 Docker 容器镜像尚未在你的系统中，默认情况下 Docker 会查看 Docker Hub Registry。

图 1-1 展示了 Docker Hub Registry 页面。

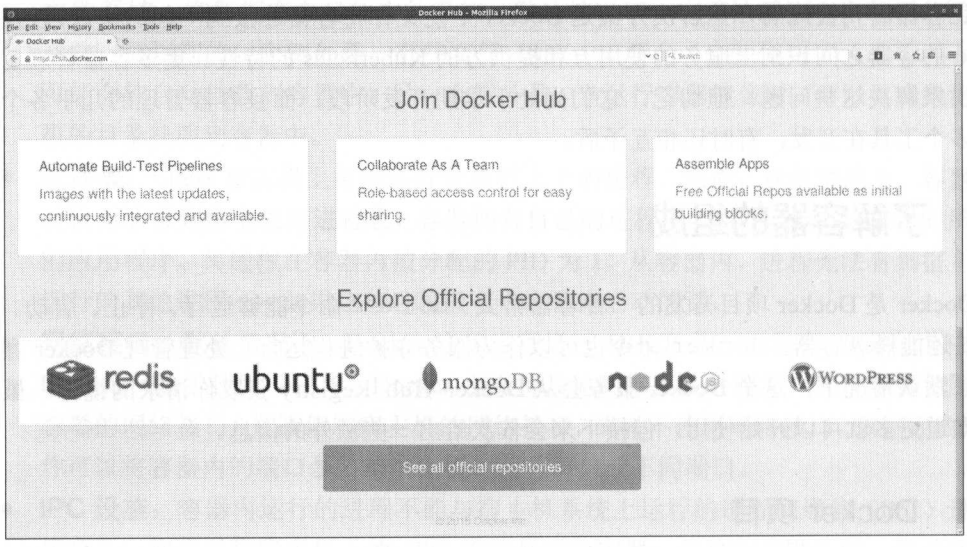


图 1-1 Docker Hub Registry 保存了数以千计的 Docker 镜像

通过注册 Docker 用户账号，你可以拥有自己的 Docker repository，你可以把 Docker 容器镜像推送到这里。之后，只要连接了因特网，你就可以从任何运行 Docker 的系统拉取这些镜像。

Linux 发行版和应用项目在 Docker Hub Registry 上有官方的 repository。除了 Docker 容器镜像本身，多数情况下，你也能在 Docker Hub Registry 上查找使用这些镜像的说明以及用来构建这些镜像的 Dockerfile 文件。对于不想公开共享的容器镜像，也有办法创建你自己的 Docker Registry，你可以用它来私密地保存镜像或者直接从 Docker 项目购买安全的容器存储空间。

你也许注意到，当描述 Docker 存储和传输软件所用的格式时，会出现“镜像”和“容器”这样的词。当使用 Docker 时，理解镜像和容器之间的差别是非常关键的。

1.2.3 Docker 镜像和容器

容器化的目的是将应用程序运行需要的所有组件集合在一个单一而独立的单元中。对于 Docker 而言，这个单元被称作 Docker 镜像。镜像之中是容器要运行的应用程序以及应用程序执行所需的库、配置文件、可执行程序或者其他组件。

镜像是一个静态单元，它要么在 repository 中，要么在安装了 Docker 的本地文件系统中，并等着被运行。与将 Docker 镜像存储到 repository 中不同，将 Docker 镜像保存到文件系统时，它会被保存为一个 tarball 文件。这个 tarball 可以像其他文件那样传输，之后可以在运行 Docker 的本地系统上被导入并作为容器运行。

像 Red Hat Enterprise Linux、Ubuntu、Fedora 和 CentOS 这样的主要 Linux 发行版都提供了官方的基础镜像，你可以使用它们构建自己的 Docker 镜像。即使不是程序员，也可以获取基础镜像、向其中添加现有应用程序并将其制作为自己的镜像。要做到这一点只需创建 Dockerfile 并在其上运行 `docker build` 命令。

Docker 容器这个术语指的是 Docker 镜像的运行实例，或者更为准确地说，是一个已运行的镜像的实例，因为它此刻可能正在运行、暂停或者已经停止。开始使用 Docker 时，镜像和容器之间的区别非常关键。需要理解这个区别的原因在于，处理镜像和处理容器有着不同的命令。

例如，当要查看本地系统的镜像列表时，要输入 `docker images`；要查看正在运行的容器列表，要输入 `docker ps`（或是 `docker ps -a`，查看不再运行但还保存在你的系统中的容器的列表）。

想从镜像运行容器，要用 `docker run` 命令。想停止正在运行的容器，要用 `docker stop` 命令。在容器停止后，可以使用 `docker start` 命令再次启动已停止的容器。只是要暂停容器中的所有进程，要用 `docker pause` 命令。之后输入 `docker unpause` 来再次启动已暂停的容器。谨记，`docker run` 从原始镜像运行一个新容器，而 `docker start` 从容器被停止时的状态重新启动一个容器（例如，你添加的软件或修改的文件还在那儿）。

当使用容器时，你会注意到所有给出的示例都有一个共同点。它们中的每一个都要使用 `docker` 命令来调用。

1.2.4 docker 命令

`docker` 命令是你与 Docker 容器和镜像直接打交道所使用的主要命令。实际上，在 Docker 软件的一些软件包中，`docker` 只是为数不多的几个可执行命令之一。

一旦按第 2 章的描述安装了 Docker 软件，只要启动 Docker 服务就能开始使用 `docker` 命令。`docker` 命令内置的一个不错的特性是 Tab 自动补全（如果你是从默认的 bash shell 运行 Docker 的话）。因此，一旦 Docker 服务运行起来，你就可以作为 root 用户输入 `docker`，紧跟着按下两次 Tab 键来查看可用的 `docker` 子命令：

```
# docker <Tab><Tab>
attach    exec      inspect   port      rmi       tag
build     export    kill      ps        run       top
commit    help      load      pull      save      unpause
cp        history   login     push      search    version
create    images    logout    rename    start     wait
diff      import    logs      restart   stats
events    info      pause     rm        stop
```

因为本书第二部分的多数章节专门讲授 docker 如何与这些子命令一起使用，所以我不会花太多时间展示它们都是如何工作的。相反，我会概述能用它们做什么。

- **查找 Docker 组件的信息**：使用 `docker version` 展示 Docker 特性的版本信息。使用 `docker info` 查看运行 Docker 的系统的信息。`docker help` 查看可以与 docker 命令一起使用的命令和选项。使用 `docker history` 展示镜像的历史。使用 `docker inspect` 查看镜像或容器的信息。使用 `docker port` 列出容器的端口映射。
- **操作正在运行的容器**：使用 `docker ps` 列出正在运行的容器。使用 `docker attach` 将另一个命令附加到正在运行的容器上。使用 `docker exec` 在正在运行的容器中执行命令。使用 `docker inspect` 审查容器的元数据。使用 `docker cp` 从容器向宿主机系统复制文件。使用 `docker diff` 检查容器从启动后其文件系统所做的改变。
- **操作镜像**：使用 `docker images` 列出系统上的镜像。使用 `docker run` 运行镜像。使用 `docker pull` 从 registry 向本地系统拉取镜像。使用 `docker push` 将镜像推送到 registry 中。使用 `docker save` 将镜像保存为 tarball。使用 `docker load` 从 tarball 文件加载本地镜像。使用 `docker export` 从容器中将文件系统导出成本地系统的 tarball 文件。
- **操作 Docker registry**：使用 `docker search` 在 registry 中搜索镜像。使用 `docker login` 登录到 Docker Hub Registry（你就能用自己的账号来推送和拉取镜像）。使用 `docker logout` 从 Docker Hub Registry 登出。
- **修改现存的镜像**：使用 `docker tag` 为镜像添加一个名字。使用 `docker rename` 修改镜像的名字。
- **修改容器的状态**：使用 `docker stop` 停止正在运行的容器。使用 `docker start` 启动一个已经停止的容器。使用 `docker pause` 暂停容器。使用 `docker unpause` 重新启动一个已经暂停的镜像。使用 `docker kill` 向容器发送 kill 信号或其他信号。使用 `docker restart` 停止并重新启动容器。
- **查看 Docker 的活动**：使用 `docker events` 查看 Docker 服务器的事件。使用 `docker top` 查看容器的进程活动。使用 `docker logs` 查看由容器产生的日志消息。使用 `docker stats` 查看容器的 CPU 和内存使用统计。使用 `docker wait` 查看容器直

到它停止，然后打印它的退出代码。

- **创建镜像和容器**：使用 `docker build` 从头构建镜像。使用 `docker commit` 从容器创建镜像。使用 `docker create` 从镜像创建容器但不运行它。使用 `docker import` 将文件系统导入镜像中。
- **删除容器和镜像**：使用 `docker rm` 删除已停止的容器。使用 `docker rmi` 删除镜像。

尽管 Docker 是专为帮你以最少的麻烦让容器运行起来而设计的，本书在这里会指导你完成 Docker 的初次体验，并指出你可能无法自己发现的有趣特性。之后，本书会带你进入 Docker 中一些很少被绘制的水域——与多容器的部署和管理相关。换句话说，本书为你提供了一种方法来进行 Docker 探险。

1.3 探究容器

由于围绕 Docker 和容器的开发节奏很快，现在任何书籍所能做的最好的就是将你带到一条正确的路上。本书中，这就意味着一开始会给出一套靠谱示例来说明当前 Docker 和精选的一组支持工具是如何工作的。之后，这条大路就会展示位于地平线上的新特性和新工具。

无论你是想使用和管理容器，还是担负容器开发的任务，本书会让你以所有人都需要的一些事情起步，包括以下几项。

- **搭建 Docker**：许多完整的 Linux 系统和一些面向容器的特定 Linux 系统上都可以使用 Docker。因此，本书开始部分的说明会帮你选择一个或多个这样的系统并展示如何启动 Docker 服务。
- **搭建 Docker registry**：Docker 把容器镜像保存在 registry 中并使镜像可以被拉取（下载）到运行 Docker 的系统中。因此，你可以学习如何创建自己的私有 Docker registry 并用它保存自己的容器镜像。

如果你想要使用和管理容器，我会为你介绍不同的方法来使用：

- 普通的单个容器；
- 一组容器（使用 Kubernetes 和其他工具来管理它们）。

无论你是创建容器化应用还是只运行容器化应用，它都有助于理解将底层操作系统特性提供给容器的方法。你应该了解的用来支持容器的操作系统的辅助特性包括以下几个。

- **宿主机权限**：设计上限制了容器可以操纵宿主机的范围。开放宿主机权限能让容器直接访问宿主机系统的特性，如宿主机的进程表、设备、特定的 CPU 以及 IPC 命名空间。本书稍后会用 Fedora Atomic Host 演示那些专为访问和修改宿主机系统而设计的容器（被称为超级特权容器）。
- **存储**：你可以在 Docker 容器中使用挂载来连接宿主机的存储空间，而不是将数据保存在容器中。
- **网络**：有特殊的规则和选项来从容器内管理宿主机的网络接口。

作为软件开发人员，容器既限制了你也让你得以自由。你有能力确保你的应用程序的所有文件都以一种立即可以运行的形式与容器打包在一起。但新的挑战浮现出来，这些挑战要求你重新思考应对下列事项的开发方式。

- 高效地处理容器层；
- 通过测试、开发和生产阶段指引软件增强；
- 在多个容器间划分服务；
- 在运行时环境中部署、启动和停止容器；
- 处理你的应用程序所需要的宿主机系统的辅助服务。

Docker 有许多强大的特性并且当前可以很好地构建和运行单个容器。但 Docker 周围的世界并非停滞不前。甚至此时，数以百计的人们每天都在努力扩展用 Docker 所能完成的工作。

相当数量的工具即将出现，用以支持企业级容器的开发和部署。同样地，人们在不断地创造聪明的容器，并将这些容器以及用于构建这些容器的 Dockerfile 提供给想要使用和扩展它们的人。为了帮你采取超出本书范围的下一步措施，最后两章描述了以下内容。

- **更多容器化工具**：如此众多的工具被开发出来使用 Docker 容器，以至于很难搞清全部这些工具。这一章描述了那些你将很快就能用于 Docker 容器的新兴工具。
- **容器镜像的示例**：为了说明人们使用 Docker 的创造性方式，我用了整整一章来展示那些已经开放出来让你尝试的很酷的容器。

紧跟着最后一章专门用来帮你评估哪些现存的和即将推出的工具在你力图使用容器扩展自己的道路时对你最为有用。它梳理了当今这些工具能够帮你做什么，以及它们正如何发展以便明天为你服务。

1.4 小结

通过将应用程序（以及应用程序运行所需要的全部东西）与运行该应用程序的宿主机解耦，Docker 容器提供了一种简单、优雅的应用程序部署和运行方式。较之于将应用程序直接安装在宿主机上所获得的灵活性，Docker 容器能够提供更大的使用灵活性。而且，与虚拟机相比，Docker 容器对系统资源（如 CPU 使用、内存消耗以及磁盘空间使用）的要求很少。

本章描述了构成 Docker 的不同组件。这些组件由 Docker 项目管理（https://www.docker.com），包括 Docker Hub Registry（它保存 Docker 镜像）、Docker 服务、`docker` 命令以及你要用到的镜像和容器。

本书将你置于这样一条道路上，一开始先了解使用 Docker 来操作单个容器的所有细节。在那里，会带你了解特殊用例，如特殊的特权容器，并向你介绍你需要了解的有关存储和网络的问题。

对于程序员而言，可以学到一些有关容器开发的最佳实践。最后，本书给出容器的示例，以便对你要亲自构建的东西以及不久的将来围绕 Docker 能设想的新开发有所启发。

第 2 章

搭建容器运行时环境

本章主要内容

- 在标准 Linux 系统中搭建 Docker
- 在容器专用的 Linux 系统中搭建 Docker

自构建之初，Docker 就运行在 Linux 之上。但与虚拟机不同，虚拟机包含了整个操作系统，而容器本身只是依靠单独的操作系统为其提供运行环境。这就是说，我们有诸多可选的方式来创建 Docker 的工作环境。具体方式包括以下这些。

- **标准 Linux**：多个主要的 Linux 发行版已将 Docker 打包在其中。如需最新的 Docker 特性，只要安装相应 Linux 发行版的最新版本即可。切记，Docker 与操作系统有着密切的联系，因此，Linux 的早期版本可能并不包含 Docker 运行所需的全部特性。本书将介绍在流行的 Linux 发行版（如 Ubuntu、Fedora 和 Red Hat 企业版）上搭建 Docker 的方法。至于其他 Linux 系统上的 Docker 安装，本书会提供相关说明的链接。
- **Microsoft Windows、Mac OS X 或其他通过虚拟机运行的方式**：Docker 不能直接在 Windows 或 Mac 系统上运行。但是，如果 Windows 或 Mac 系统的版本能够运行虚拟机，并且硬件也足以支撑虚拟机的运行，就能够安装 Linux 虚拟机并通过它来运行 Docker。但要注意，如果计算机的处理能力和 RAM 不够强劲，那么这种运行 Docker 的方式体验并不是最好的。但是，如果你拥有最新的 Windows 或 Mac 系统并且硬件够用，Docker 应该可以很好地运行。
- **容器专用 Linux**：运行容器并不需要完整的桌面或服务器 Linux 系统。实际上，容器原本就是为了容纳与其一同运行所需的软件，因此底层的 Linux 系统可以非常轻量级。像 Project Atomic 和 CoreOS 这样的 Linux 发行版尤其适合为容器提供运行时环境。你可能想在容器开发阶段使用标准 Linux，这样可以非常容易获取所需的开发工具。之后使用 Project Atomic 或 CoreOS 来部署这些容器——要么直接部署在硬件上，要么部署在亚马逊的 EC2 或谷歌云平台（Google Cloud Platform）这样的云环境中。

本章将介绍如何按刚刚描述的那样准备计算机系统以运行 Docker 容器。一旦系统准备就

绪，就可以跟随本书之后的示例进行容器的运行、管理、部署和编排。

2.1 标准 Linux 系统的 Docker 配置

如果刚刚开始接触 Linux，建议安装一个带桌面的标准 Linux 发行版（带桌面的服务器、桌面系统或者是工作站）。如此，在创建和运行容器的过程中，就有一整套开发、调试和监控的工具。当准备长久运行容器时，可以考虑使用 Project Atomic 或 CoreOS 来部署容器。

在 Linux 中使用 Docker，首要的是包含了 `docker` 命令的软件包。在 Ubuntu 中，包含 `docker` 命令的软件包叫 docker.io。在 Fedora 中，这个软件包之前叫 docker-io，这是为了避免与另一个称为 docker 的软件包冲突——那个软件包提供了应用程序在桌面停驻的特性。目前，最新的 Fedora 已将包含 `docker` 命令的软件包命名为 docker。在 Red Hat 企业版中，Docker 容器的软件包就直接称为 docker。

2.1.1 Ubuntu 的 Docker 配置

在 Linux 爱好者中，Ubuntu（http://www.ubuntu.com/）是一款流行的 Linux 发行版。除提供标准桌面系统外，Ubuntu 也有轻量级桌面系统（如 Xubuntu 和 Lubuntu）以及专门用于教育和多媒体的系统。虽然 Ubuntu 大约每 6 个月就会发布新版本，其中多数只会得到 9 个月的支持，但有些发布是长期支持（Long Term Support，LTS）版本，它们将得到 5 年的支持。

为了在 Ubuntu 中使用 Docker，要从 Ubuntu 最新的 LTS 版本着手。我使用的是 Ubuntu 14.04 桌面版（最新的 LTS 版本）。请访问如下链接，获取 Ubuntu14.04 桌面系统的 live 版或安装媒介以及相关说明。

- **下载 Ubuntu**。访问 www.ubuntu.com/download/destop，下载与你的计算机匹配的 ISO 映像文件。对于多数新计算机，应该使用 64 位 PC 版，而对于比较老的计算机或者配置低的计算机，请下载 32 位的 ISO 映像文件。
- **安装 Ubuntu**。参考“安装 Ubuntu 14.04 LTS”（www.ubuntu.com/download/desktop/install-ubuntu-desktop）中的说明来安装该版本的 Ubuntu。

1. 在 Ubuntu 中安装 Docker（docker.io 软件包）

一旦安装好 Ubuntu 并准备继续，请以系统所有者的身份登录系统，并通过以下步骤让 docker.io 服务运行起来。

（1）**更新软件包列表**。在开始安装 Docker 前，需要先更新 Ubuntu 的软件包列表。

```
$ sudo apt-get update
...
Reading package lists... Done
```

（2）**安装 docker.io 软件包**。在 Ubuntu 中，Docker 是由 docker.io 软件包提供。安装该

软件包时，其依赖的软件包也会同时安装：

```
$ sudo apt-get install docker.io
```

（3）**启动 docker.io 服务**。Docker 服务应该是自动启动的。为了确认其已启动，键入下面的命令：

```
$ sudo service docker.io status
docker.io start/running, process 1236
```

输出显示 docker.io 服务已经运行。接下来要做的是研究一下 docker.io 软件包的内容。

2. 探查 Ubuntu 的 docker.io 软件包

为了对构成 Docker 的组件有切身的体会，我们先看看 docker.io 软件包的内容。使用下面的命令列出 docker.io 软件包的内容：

```
$ sudo dpkg-query -L docker.io | less
/usr
/usr/share
/usr/share/man
/usr/share/man/man1
/usr/share/man/man1/docker.io.1.gz
/usr/share/docker.io
/usr/share/docker.io/contrib
/usr/share/docker.io/contrib/mkimage-alpine.sh
...
/usr/share/doc
/usr/share/doc/docker.io
/usr/share/doc/docker.io/README.Debian
/usr/share/zsh
/usr/share/zsh/vendor-completions
/usr/share/zsh/vendor-completions/_docker.io
/usr/bin
/usr/bin/docker
/usr/lib
/usr/lib/docker.io
/usr/lib/docker.io/dockerinit
/etc
/etc/bash_completion.d
/etc/bash_completion.d/docker.io
/etc/init
/etc/init/docker.io.conf
/etc/init.d
/etc/init.d/docker.io
/etc/default
/etc/default/docker.io
/lib
/lib/udev
/lib/udev/rules.d
```

```
/lib/udev/rules.d/80-docker.io.rules docker-io
/lib/systemd
/lib/systemd/system
/lib/systemd/system/docker.io.service
/usr/bin/docker.io
```

docker.io 软件包自带的文档中包含了一个描述 docker 命令的联机帮助页（键入 man docker 来查看它）以及 README 文件，这些文件位于/usr/share/doc/docker.io 目录中。到读者阅读本书时，docker.io 软件包应该会包含更多的联机帮助页。

/usr/share/docker.io/contrib 目录下存放了各种脚本，有些可以用来创建最小的文件系统，而另一些则可以完成其他任务，如创建基础 Docker 映像文件。这个目录下的各个子目录中存放了用于构建、导入和运行 Docker 映像的 Dockerfile 文件。

docker 命令提供了 Tab 命令补全。用于 bash 的 Tab 命令补全文件（/etc/bash_completion.d/ docker）和 zsh 的 Tab 命令补全文件（/usr/share/zsh/vendor-completions/_docker.io）也来自 docker.io 软件包。

docker.io 软件包提供的主要命令就是 docker（/usr/bin/docker）。docker 命令用来创建、操作和维护 Docker 映像文件和容器。这个 docker 命令还作为守护进程运行，来提供 Docker 服务。

对于 Docker 服务，docker.io 软件包中剩余的多数文件与 Docker 服务的启动方式相关。无论你的系统是使用 Upstart（init）还是 systemd 来初始化操作系统，对应的 Docker 服务启动文件都已包含在 docker.io 软件包中。

目录/var/lib/docker 并不是 docker.io 软件包的一部分，但在安装 Docker 时会创建该目录。请密切关注该目录。当创建 docker 映像文件和容器时，它们会被保存在该目录结构中，因此，要确保不要耗尽其所在的磁盘空间。

此时，Ubuntu 已经为使用 Docker 做好了准备。如果想开始使用 Docker，可以直接跳到第 3 章。

2.1.2 Fedora 的 Docker 配置

Fedora（https://getfedora.org）是 Red Hat 公司赞助的免费、前沿的 Linux 发行版。Fedora 大约每 6 个月发布一次新版本，其被当作新软件的实验场。

许多人在工作中使用 Red Hat 企业版 Linux，他们会在自己的笔记本或家里的台式机上安装 Fedora。这是因为 Fedora 不仅提供了软件开发工具，以及一系列服务器和系统管理软件，还提供了大量桌面工具和有趣的东西（如游戏）——那些你会为自己的个人计算机安装的软件。

如果打算在 Fedora 上使用 Docker，建议安装最新的 Fedora。我就下载并安装了 Fedora workstation live ISO。Docker 并不需要桌面系统，因此可以只安装一个基本系统，然后完全使用命令行来工作。

要在 Fedora 中使用 Docker，需要先安装 Fedora 和 docker 软件包。安装完成后，就能够启动 Docker 服务，并开始获取 Docker 映像文件，构建自己的映像文件，以及管理由这些映像文件生成的容器。

要获取最新的 Fedora Workstation live 版或安装媒介，以及相关说明，访问如下链接。

- **下载 Fedora**。访问 https://getfedora.org/，选择 Workstation。点击“马上下载”（Download Now）按钮，之后选择 64 位 Fedora Workstation 安装介质并点击“下载”（Download）按钮开始下载。（如果你的计算机是 32 位的，或者你需要不同类型的安装，如服务器或云，请选择其他映像文件。）
- **安装 Fedora**。参考安装指南来帮助安装 Fedora（如果有最新的 Fedora 可用，请使用相应的安装指南）：http://docs.fedoraproject.org/en-US/Fedora/22/html/Installation_Guide/。

1. 在 Fedora 中安装 Docker（docker 软件包）

Fedora 安装好后，登录系统。使用 root 用户或 sudo 来运行下面的命令。在安装 Docker 前，首先作为 root 用户更新 Fedora 软件（yum update），然后使用 yum install 来安装 docker 软件包（在早前的 Fedora 中，是 docker.io 软件包），并按如下方法启动 Docker 服务（另一个选择是，在最新的 Fedora 发行版中，可以使用 dnf 命令代替 yum 来安装软件）。

（1）**更新软件包**。运行下面的命令来安装最新的 Fedora 软件：

```
# yum update
```

（2）**安装 docker 软件包**。早期版本的 Fedora 中，Docker 是由 docker.io 软件包提供的。现在，这个软件包可能已经叫 docker 了。按下面的方法安装该软件包，其依赖的软件包也会一并安装：

```
# yum install docker
```

（3）**启动 Docker 服务**。Fedora 并不会自动启动 Docker 服务。输入下面的命令，启用 Docker 服务并立即启动它：

```
# systemctl enable docker.service
Created symlink from /etc/systemd/system/multi-user.target.wants/ docker.
service to /usr/lib/systemd/system/docker.service.
# systemctl start docker.service
```

（4）**检查 Docker 服务的状态**。输入如下命令，确保 Docker 服务已运行：

```
# systemctl status docker.service
  docker.service - Docker Application Container Engine
    Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled)
    Active: active (running) since Sun 2015-05-17 10:05:32 EDT; 2min ago
      Docs: http://docs.docker.com
  Main PID: 3405 (docker)
    CGroup: /system.slice/docker.service
```

```
└─3405 /usr/bin/docker -d --selinux-enabled
```

输出显示 docker.service 已经启用并处于活动状态。实际的服务由 docker 命令组成（/usr/bin/docker），以守护进程的方式运行（-d），并且启用了 SELinux（--selinux-enabled）。接下来，看一看 docker 软件包的内容。

2. 探查 Fedora 的 docker 软件包

安装 docker 后，在 docker 软件包（或 docker.io 软件包）上执行 rpm -ql 命令来查看该软件包的内容。这能让你初步了解如何使用 Docker：

```
# rpm -ql docker | less
/etc/docker
/etc/sysconfig/docker
/etc/sysconfig/docker-network
/etc/sysconfig/docker-storage
/etc/udev/rules.d/80-docker.rules
/usr/bin/docker
/usr/bin/docker-storage-setup
/usr/lib/systemd/system/docker.service
/usr/libexec/docker
/usr/libexec/docker/dockerinit
/usr/share/bash-completion/completions/docker
...
/usr/share/doc/docker/README.md
/usr/share/man/man1/docker-attach.1.gz
/usr/share/man/man1/docker-build.1.gz
/usr/share/man/man1/docker-commit.1.gz
...
```

多数文件与 Ubuntu 的 docker.io 软件包中的文件相同。有些文件用来启动 Docker 服务（最新的 Fedora 发行版仅使用 systemd），也有 docker 命令自身的文件（/usr/bin/docker）。

同样，docker 软件包提供了一个用于 Tab 命令补全的文件，当在 bash shell 中使用 docker 命令时，它可以自动补全所输入的命令。能够补全的内容包括 docker 的子命令和选项、容器和镜像的名字及 ID，以及镜像库的名字和标签。

下面介绍 docker 软件包中一些特定于 Fedora 系统的文件。

- /etc/sysconfig/docker：当 docker 命令作为服务的守护进程来运行时，这个文件用来收集提供给 docker 命令的选项。默认情况下，仅增加--selinux-enabled 选项来启用针对 Docker 的 SELinux 支持。该文件中的一项配置（DOCKER_CERT_PATH）将启动 Docker 服务所需的密钥文件存放目录指定到/etc/docker。
- /etc/sysconfi/docker-network：该文件用来收集传给 Docker 服务的网络选项。
- /etc/sysconfig/docker-storage：该文件用来改变 Docker 服务存储数据的方式。默认会使用挂载到/var/lib/docker 目录的一个稀疏的回送（loopback）文件。可以选择使用原始存储设备来保存元数据和数据。请阅读该文件来了解如何进行设置。按

这种方式使用原始存储设备不但能够提高性能，而且可以摆脱回送文件 100 GB 的最大存储限制。

- /etc/udev/rules.d/80-docker.rules：该文件用来告知 udev 服务配置 Docker 服务所使用的设备映射文件，Docker 服务使用该设备映射文件来访问容器所需的宿主机特性。（Ubuntu 的 Docker 软件包含有类似的 udev 文件。）
- /usr/share/doc：在`/usr/share/doc/docker`目录中存放有来自 Docker 项目的文档。这既有软件变更和软件许可方面的信息，也有关于项目自身的 README 文件。
- /usr/share/man 文档：`docker`命令的联机帮助文档分散在多个 docker 页中，这些联机帮助文档按子命令组织，存储在第 1 部分（man1）。例如，要查看`docker`命令的`build`选项（`docker build`），只要输入`man docker-build`命令即可。第 5 部分只有一个联机帮助页，它描述了用来构建 Docker 镜像的 Dockerfile 的结构。（输入`man Dockerfile`来查看该文件。）

docker 软件包已安装就位，可以直接跳到第 3 章开始使用`docker`命令。

2.1.3 Red Hat Enterprise Linux 的 Docker 配置

Red Hat Enterprise Linux（RHEL）是由 Red Hat 制作的基于订阅的 Linux 发行版，其旨在提供一个经过测试的、稳定的 Linux 发行版，并结合了来自 Red Hat 的不同级别的客户支持。在安全性和稳定性至关重要的情况下，整个应用程序开发生命周期中的系统通常都会使用 Red Hat Enterprise Linux，这也包括生产环境的部署。

与 Fedora 类似，Red Hat Enterprise Linux 是基于 RPM 的，但安装 Red Hat Enterprise Linux 和 Docker 时有些步骤与 Fedora 不同。

在下载和安装 Red Hat Enterprise Linux 之前，要确保你拥有有效的订阅或已报名获取了用于评估的订阅。访问下面的链接来了解如何下载和安装 RHEL。

- 下载 RHEL。访问 https://access.redhat.com/downloads/，点击 Red Hat Enterprise Linux 最右侧的按钮，并在弹出的菜单中选择 Download Latest（Server），或者点击 Red Hat Enterprise Linux 链接并选择适合你的安装介质。如果网站提示你登录，而你尚未获得 Red Hat 的订阅，请点击 Evaluation（评估）按钮来报名获取评估订阅。要注意的是，你需要软件的 RHEL 服务器版，因为其他版本 RHEL 不包含 Docker。
- 安装 RHEL。访问 Red Hat Enterprise Linux 文档页面（https://access.redhat.com/documentation/en/red-hat-enterprise-linux/），选择 Installation Guide（安装指南）。安装指南介绍了如何启动刚下载的安装媒介，以及各种不同的软件安装方法。

一旦安装好 RHEL，需要注册系统、安装 docker，并启动 Docker 服务，具体描述如下。

1. 在 Red Hat Enterprise Linux 中安装 Docker（docker 软件包）

假设你已有 Red Had Enterprise Linux 的订阅，并有 Red Hat 客户门户（customer portal）的账号，按照下列步骤就能订阅系统并为其添加 docker。

（1）**注册和订阅 RHEL**。使用订阅管理器（subscription manager）来启用 Red Hat 订阅和获取 docker 软件包所要用的软件库，以及之后要用到的其他相关软件包。执行下列命令，注意替换为你的 Red Hat 用户名，并（在提示时）输入密码：

```
# subscription-manager register --username=rhnuser --auto-attach
Password:
# subscription-manager repos --enable=rhel-7-server-extras-rpms
# subscription-manager repos --enable=rhel-7-server-optional-rpms
```

（2）**更新软件包**。RHEL 7 系统取得适当的资格后，运行如下命令来将所有已安装的软件包更新到最新版，并重启系统：

```
# yum update
# reboot
```

（3）**安装 docker**。系统重启后，键入下面的命令来安装 docker 软件包：

```
# yum install docker
```

（4）**启动并启用 Docker 服务**。Docker 安装后，只要启动并启用 Docker 服务就可以开始使用 Docker 了。在 RHEL 7 中，输入下面的命令来启动和启用 Docker 服务：

```
# systemctl start docker.service
# systemctl enable docker.service
# systemctl status docker.service
docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled)
   Active: active (running) since Sun 2015-05-17 16:52:33 EDT; 8s ago
     Docs: http://docs.docker.com
 Main PID: 32147 (docker)
   CGroup: /system.slice/docker.service
           └─32147 /usr/bin/docker -d --selinux-enabled \
                   --add-registry registry.access.redhat.com
```

通过这个输出，可以看到 Docker 服务已启用并处于活动状态。提供服务的 `docker` 命令（`/usr/bin/docker`）启动时接收了这些参数，作为守护进程启动执行（`-d`），启用 SELinux，并添加了 docker 用来搜索的容器 registry（这里是 `registry.access.redhat.com`）。

现在，docker 软件包已经安装好，Docker 服务也已经运行，接下来熟悉一下 RHEL 的 docker 软件包的内容。

2. 探查 RHEL 的 docker 软件包

运行下面的命令查看 docker 软件包的内容：

```
# rpm -ql docker
/etc/docker
```

```
/etc/docker/certs.d
/etc/docker/certs.d/redhat.com
/etc/docker/certs.d/redhat.com/redhat-ca.crt
/etc/docker/certs.d/redhat.io
/etc/docker/certs.d/redhat.io/redhat-ca.crt
...
/usr/share/man/man1/docker-attach.1.gz
/usr/share/man/man1/docker-build.1.gz
/usr/share/man/man1/docker-commit.1.gz
...
/usr/share/man/man5/Dockerfile.5.gz
/usr/share/rhel
/usr/share/rhel/secrets
/usr/share/rhel/secrets/etc-pki-entitlement
/usr/share/rhel/secrets/rhel7.repo
/usr/share/rhel/secrets/rhsm
...
/usr/share/vim/vimfiles/doc
/usr/share/vim/vimfiles/doc/dockerfile.txt
/usr/share/vim/vimfiles/ftdetect
/usr/share/vim/vimfiles/ftdetect/dockerfile.vim
```

RHEL 的 docker 软件包与 Fedora 的 docker 软件包几乎完全相同，因此，可以参考之前在 Fedora 中安装 docker 软件包的内容，来了解这些组件的描述。这里只描述了特定于 RHEL 的 docker 软件包组件。

- **Certificates**：RHEL 的 docker 软件包已做过一些配置，它并未使用 Docker Hub 作为其 registry。实际上，对于官方的 RHEL 容器镜像，Red Hat 使用了自己的 registry。目录`/etc/docker/certs.d`中保存了 Docker 服务使用的证书，Docker 服务使用这些证书来确保其在与 Red Hat 的 registry 进行通信。
- **Secrets**：目录`/usr/share/rhel/secrets`中存放的文件将会复制到系统运行的容器中。特别是，RHEL 的 docker 软件包会将容器使用 Red Hat 的订阅管理所需要的文件放在 secrets 目录下。这些与订阅管理相关的文件使容器可以使用宿主机的订阅权限来安装和管理容器自己的 RPM 包，而无需消耗额外的订阅。
- **vimfiles**：当使用 `vim` 命令来编辑 Dockerfile 时，目录`/usr/share/vim/vimfiles/`中的文件提供了语法高亮。这有助于 Dockerfile 文件的创建和编辑。

2.1.4 其他操作系统的 Docker 配置

之前提到过，Docker 不仅仅可以用于上面介绍的这些操作系统。如需在其他操作系统或云环境中安装 Docker，参考下面的 Docker 安装过程说明。

- **在 Mac OS X 上安装 Docker**。请使用 Mac OS X 10.6 或更新的版本，如下安装过程使用了包含 Boot2Docker 工具（https://github.com/boot2docker/boot2docker/releases）的 ISO 镜像，通过该工具，其在 Mac 上创建了一个运行在 VirtualBox 中的轻量级 Linux

虚拟机：

```
https://docs.docker.com/installation/mac/
```

- **在 Microsoft Windows 上安装 Docker**。对于使用 Windows 7.1 或 8（以及其他版本）的系统，如果你的计算机支持硬件虚拟化，可以使用下面链接给出的安装过程。重申一下，Docker 只能在 Linux 中运行，因此这个安装过程要使用 Boot2Docker 工具创建一个 Linux 虚拟机，而你可以用这个虚拟机来尝试 Docker：

```
https://docs.docker.com/installation/windows/
```

- **在 CentOS 上安装 Docker**。CentOS 上运行 Docker 与 Red Hat Enterprise Linux 差不多，但无需支付订阅费用。需要注意的是，CentOS 并未提供任何保障，而且会在特性和安全补丁方面落后于 RHEL。参考下面的链接在 CentOS 上安装 Docker：

```
https://docs.docker.com/installation/centos/
```

- **在 Debian 上安装 Docker**。参考下面的说明在 Debian GNU/Linux 系统上安装 Docker：

```
https://docs.docker.com/installation/debian/
```

- **在 SUSE 上安装 Docker**。下面的说明描述了在 SUSE 上安装 Docker 的过程，既包括 openSUSE，也包括 SUSE Linux Enterprise：

```
https://docs.docker.com/installation/SUSE/
```

如果这里没有涉及你使用的操作系统，请访问 Docker 的安装页面（https://docs.docker.com/installation/），了解在其他操作系统上安装 Docker 的相关说明。如果你打算亲自动手为自己的操作系统组装 Docker，请参考 Docker 的二进制文档页面（https://docs.docker.com/installation/binaries/）。

还有一类 Linux 系统的 Docker 使用尚未涉及，这就是专为运行容器而设计的 Linux 系统。接下来将介绍如何搭建 Project Atomic 和 CoreOS 操作系统，使其提供一个容器专用的运行时环境。

2.2 容器式 Linux 系统的 Docker 配置

我们使应用程序容器化，不遗余力地使其变小和变高效，但如果最终要将这些容器部署到缓慢、臃肿的操作系统中，那将使一切努力失去意义。在不断演进的容器模型中，既然容器已包含运行应用程序所需的可执行文件、库以及其他组件，宿主操作系统完全可以简化到只保留运行容器所需的功能。

Project Atomic 和 CoreOS 这两个项目的目标是提供专为运行容器而优化的操作系统。这

样的操作系统既能够直接运行在硬件上，也能运行在公有云（如亚马逊的 EC2 或者 Google Compute Engine）、（OpenStack、VMware 和 Red Hat Engerprise Virtualization 提供的）私有虚拟化平台上或者（像 Linux KVM 宿主机这样的）个人计算机上的虚拟机中。

同样，安装和配置 Project Atomic 和 CoreOS 操作系统的方法很多。Project Atomic 提供了一个 ISO 的安装程序，这类似于 Fedora 和 RHEL 系统的安装——跟随安装步骤一路点击来配置网路、磁盘分区和添加用户。而 CoreOS 提供的 ISO 文件只是把整个镜像复制到所选的分区上，然后几分钟内就可以启动了。

当部署到云环境中时，像 cloud-config（CoreOS）和 cloud-init（Atomic 和其他系统）这样的工具，可以在需要时用来配置云端镜像。

2.2.1 Atomic Host 的 Docker 配置

Atomic 项目（www.projectatomic.io）是基于 RPM 的 Liunx 发行版构建器，它是为部署和管理容器而专门设计的。Fedora、Red Hat Enterprise Linux 和 CentOS 都有可以作为 Atomic 宿主机运行的版本。

Atomic Host 系统提供了多种形式，以方便在不同环境中应用。可以下载 Atomic qcow2 镜像，并通过 cloud-init 注入配置信息（用户账号、主机名、配置文件等）的方式来配置它。对于 CentOS Atomic，有用于快速启动 CentOS Atomic 虚拟机的 Vagrant 文件可用。而 RHEL Atomic 和 Fedora Atomic 项目提供了用于安装的 ISO，可以使用传统的安装程序一步一步地配置 Atomic 系统。

通过如下链接可以找到并下载不同 Atomic 宿主机版本的镜像。

- Fedora Atomic（https://getfedora.org/cloud/download）：在下载 Fedora Cloud 页面中有一个 Atomic Host Images 的标签（点击进入）。选择 Atomic Host Image 来下载镜像，并使用通常的 Fedora 安装程序（anaconda）来安装 Fedora Atomic 系统。这个页面上还有其他形式的 Fedora Atomic 可供下载，如 qcow2 格式（用于 OpenStack）以及原始格式的 Atomic 镜像文件，还可以通过 Vagrant 在 VirtualBox 和 KVM（libvirt）环境中启动 Atomic 镜像。最后，还有用于 Amazon EC2 部署的 Atomic 镜像。
- CentOS Atomic（http://buildlogs.centos.org/centos/7/isos/x86_64/）：该网站有 qcow2 格式的镜像可用。同样，如果系统已经安装了 vagrant（https://www.vagrantup.com/），这里也有 vagrant box，可以使用它们立即启动 CentOS Atomic 虚拟机。CentOS Atomic 的安装过程与 Fedora 和 RHEL 的 qcow2 格式镜像的部署过程类似。
- RHEL Atomic（https://access.redhat.com/downloads）：下载 RHEL Atomic 镜像需要 Red Hat 的订阅（正式或评估）。在 Red Hat 下载页面，选择 Red Hat Enterprise Linux，然后在 Product Variant 部分，选择 Red Hat Enterprise Linux Atomic host。针对不同平台（云、Red Hat 企业虚拟化、Microsoft Hyper-V 和 VMware vSphere），RHEL Atomic 镜像提供了多种不同格式（qcow2、ova、vhd 和 ISO）

想了解更多 Project Atomic 发行版的获取和安装方法，请参考 Project Atomic 快速入门指

南（www.projectatomic.io/docs/quickstart）。接下来的示例中，我使用 Fedora 22 安装程序搭建起一个 Fedora Atomic Host。相同的基本步骤可以用来指导通过 Atomic ISO 安装文件配置 RHEL。

1. Fedora Atomic Host 的 Docker 配置

与 CoreOS 一样，Atomic Host 系统已为运行 Docker 做好了配置。开始按这个过程搭建 Fedora Atomic Host 前，先下载 Fedora Atomic ISO 安装镜像。可以将 ISO 镜像刻录到一张 CD 或 DVD 盘上，而后可以将它直接安装在计算机硬件上，或者在几乎所有支持虚拟化的系统上使用。例如，在 Linux KVM 这样的系统中，通过 virt-manager 工具来使用。

（1）**下载 Fedora Atomic ISO 镜像**。下载最新的 Fedora Atomic 安装镜像。查找官方的 Fedora 22 或更新版本。我使用下列镜像文件：

```
https://download.fedoraproject.org/pub/fedora/linux/releases/22/Cloud_Atomic/
x86_64/iso/Fedora-Cloud_Atomic-x86_64-22.iso
```

（2）**启动 Fedora Atomic ISO 镜像**。要么从刻录到光盘上的镜像直接启动安装程序，要么使用虚拟机管理工具（如 virt-manager、OpenStack 或其他工具）作为虚拟机启动。我建议准备至少 10 GB 的磁盘分区给 Fedora Atomic，试试看。

（3）**做出安装选择**。跟随安装界面一步一步地进行设置，这些安装界面提供了如何配置系统的信息。多数特性的配置是通过 Installation Summary（安装摘要）界面完成的，如图 2-1 所示。在 Installation Summary 界面上，完成如下设置。

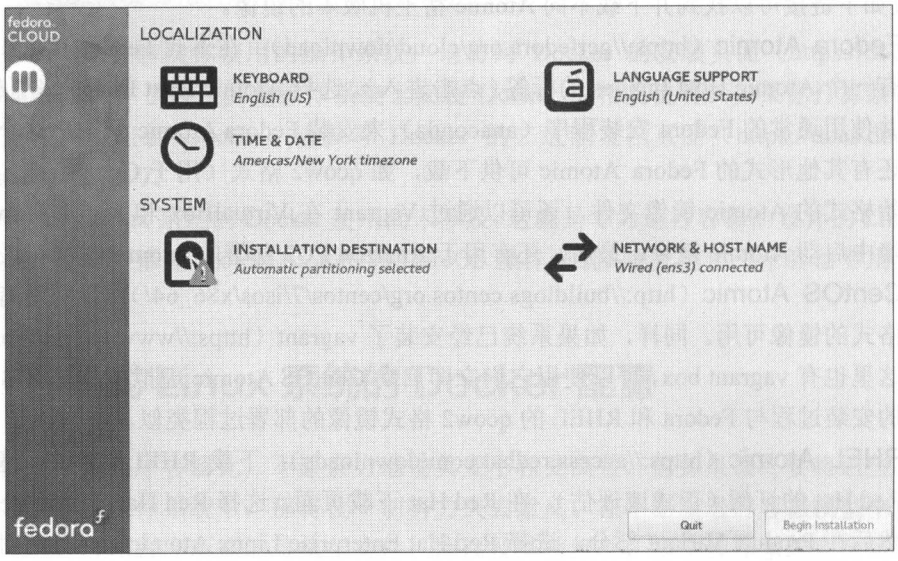


图 2-1 通过 Installation Summary 界面确定磁盘分区和网络设置

- Keyboard（键盘）：根据语言与地区选择键盘。

- Time & Date（时间和日期）：设定日期、时间和时区。默认选择的是 Network Time（网络时间），系统会和 NTP 时间服务器同步时间。
- Installation Destination（安装位置）：对于该配置项，要么采用自动分配的磁盘分区，要么自己选择一个分区。
- Language Support（语言支持）：选择语言。
- Network & Host Name（网络与主机名）：网络是自动打开的（尝试使用 DHCP 获取地址信息）。如果要关闭网络或者设置自己的主机名，选择该配置项。完成后点击 Done（完成）。

（4）**选择 Begin Installation**（开始安装）。到目前为止，还没有任何设置作用到你的磁盘分区。如果所有设置都已经完成，选择 Begin Installation。

（5）**设置 root 密码和用户**。随着安装进行，选择 ROOT PASSWORD 并设置 root 密码。之后，选择 USER CREATION 并添加用户账号（同时为该用户设置密码）。可以设置该用户是否拥有 sudo 权限（选择 Make This User Administrator）。选择 Done，返回配置界面并等待安装完成。

（6）**重启**。安装完成后，选择重启系统，现在就可以开始使用 Fedora Atomic Host 了。

2. 检查 Fedora Atomic Host

用户无需安装任何软件就可以开始使用 Docker。实际上，根本无法通过传统的 `rpm` 和 `yum` 软件包工具来为 Atomic Host 添加软件。如果要向系统添加软件，要么添加普通容器（来运行应用程序），要么添加拥有超级特权的容器（来添加直接访问宿主机系统的工具）。

重启 Fedora Atomic Host 后，就可以立即使用 Docker。docker 软件包应该已经安装，而且 Docker 服务已经启动并启用了。

通过自动升级来更新 Fedora Atomic 系统，而自动升级要使用 `atomic` 命令。我建议运行下面的命令来确保你的系统安装了最新版的 Docker 以及相关软件：

```
# atomic host upgrade
```

软件更新后，通过重启使最新的 atomic 生效。现在，可以按第 3 章所描述的搭建 Docker registry，或者开始使用 `docker` 命令——第 4 章涉及这一方面。

2.2.2 CoreOS 的 Docker 配置

CoreOS 提供了几种方式来运行 CoreOS 系统。这个例子使用了 CoreOS 的 live 镜像。这个镜像文件可以刻录到 CD 中并直接从硬件启动，或者加载到能够安装虚拟机的工具中（例如，许多 Linux 系统都有的 Virtual Machine Manager 工具）。

接下来描述了如何使用 CoreOS 的 ISO live 镜像来启动 CoreOS 的 live 系统，以及如何将 CoreOS 安装到硬盘并从硬盘启动系统，这样就可以开始使用 Docker 了。

通过以下步骤来获取 CoreOS 的 ISO live 镜像文件，并从该镜像文件启动和安装 CoreOS。

（1）**获取 CoreOS**。访问下面的链接，然后选择 Download Stable ISO 来获取 CoreOS 的安装媒介：

```
https://coreos.com/docs/running-coreos/platforms/iso/
```

（2）**准备安装媒介**。使 CoreOS 的 live 安装媒介可以将 CoreOS 安装到硬盘上。可以将它刻录到 CD 上，也可以通过 VM 安装应用程序指向它（如 Linux 的 Virtual Machine Manager），或者将它复制到 PXE 服务器上并通过网络来启动它。

（3）**启动安装媒介**。如果是安装到本地虚拟机上或者是云环境中，要确定为新 CoreOS 系统保留多少磁盘空间。当 CoreOS 启动时，磁盘将映射为/dev/sda 或/dev/sdb 这样的设备。

（4）**使用 live CoreOS**。CoreOS 会直接启动到 core 用户（该用户有 sudo 权限）的 shell 提示符下。CoreOS 默认配置了两个用户：root 和 core。目前 CoreOS 系统运行在 live 模式下（还没安装）。接下来为安装做准备。

（5）**创建加密口令**。为 core 用户创建加密口令。例如，可以使用 openssl 命令创建一个 md5crypt 口令（CoreOS 自带这个工具）。提示符出现后，输入要加密的口令（两次）：

```
# openssl passwd -1
Password: ********
Verifying - Password: ********
$1$f6e4jyo9$lbch8VJ23oU2cW5grkk4s.
```

（6）**创建 cloud-config 文件**。创建 cloud-config 文件，并通过该文件为 CoreOS 添加用户口令和其他配置信息。参考“使用 Cloud-Config”（https://github.com/coreos/coreos-cloudinit/blob/master/Documentation/ cloud-config.md）了解 cloud-config 文件内容的详细信息。在这个例子中，创建名为 cloud-config.yaml 的文件，其文件内容如下（将 passwd 的字符串替换为上一步生成的字符串）：

```
#cloud-config
users:
  - name: core
    passwd: $1$f6e4jyo9$lbch8VJ23oU2cW5grkk4s.
    groups:
      - sudo
      - docker
```

（7）**运行 coreos-install 脚本**。运行下面的命令，将 CoreOS 镜像安装到之前选定的磁盘设备上（这里使用的是/dev/sda）：

```
$ sudo coreos-install -d /dev/sda -c cloud-config.yaml
Checking availability of "local-file"
Fetching user-data from datasource of type "local-file"
Downloading the signature for http://stable.release.core-os.net...
...
Downloading, writing and verifying coreos_production_image.bin.bz2...
```

```
...
Success! CoreOS stable 647.0.0 is installed on /dev/sda
```

（8）**重启**。镜像安装完成后，重启系统：

```
$ sudo reboot
```

（9）**登录**。CoreOS 重启后，使用 core 用户以及之前为该用户分配的加密口令登录系统。至此，你就可以搭建 Docker registry（第 3 章），或者开始使用 `docker` 命令（第 4 章）。

2.3 小结

多种不同的 Linux 系统都提供了 Docker。除了使用标准 Linux 系统（如 Ubuntu、Fedora、Red Hat Enterprise Linux 或者 CentOS）来运行 Docker 外，还有专门用于容器部署的特定 Linux 系统。通过虚拟机运行这些系统，也可以在 Microsoft Windows 或者 Mac OS X 上使用 Docker。

本章介绍了配置 Ubuntu、Fedora 和 Red Hat Enterprise Linux 系统的方法，以便为运行 Docker 做好准备。本章还介绍了获取和配置特定 Linux 发型版的方法，如 Atomic Project 和 CoreOS，这样就能用它们来部署 Docker 了。

第 3 章

搭建私有 Docker Registry

本章主要内容

- 在 Fedora 或 Ubuntu 中创建私有 Docker registry
- 使用 docker-registry 软件包
- 使用 registry 的容器镜像
- 了解 Docker 镜像命名空间

当 Docker 请求使用一个现有容器镜像时，如果该镜像尚未在系统中，Docker 会从“某处”抓取它并将其下载到系统中，这种能力是 Docker 的基础之一。默认情况下，“某处”是 Docker Hub Registry（https://hub.docker.com）。然而，也有办法将获取 docker 镜像的地方配置到其他位置。这些位置被称为 registry。

通过搭建私有 registry，你可以保持自己的私有镜像。此外，还能够通过推送和拉取本地镜像来节省时间，而不用跨越因特网来完成这些工作。

搭建私有 registry 很简单，只需要（通过安装软件包或者使用 registry 的 Docker 容器镜像）获取服务，启动服务，并确保打开适当的端口，以便服务可以被访问到。与搭建 registry 相比，使用 registry 需要做更多的解释，尤其是在每天都有新特性加入 Docker，而这些特性正在不断改变 Docker 使用和查找 registry 的方式的时候。

特别是，Docker 使用镜像命名空间的方式正变得更具适应性。如果没有连接互联网——无法访问 Docker Hub，正在开发的特性允许使用一个不同的默认 registry。同样，新特性可以让你按自己的搜索顺序添加 registry，这和互联网浏览器查看不同的 DNS 服务器大致相同。

本章会介绍如何在不同的 Linux 系统上搭建私有 Docker registry。最初的例子仅仅是让 Docker registry 快速运行起来，以便开始测试和学习如何使用 registry。之后会介绍让 Docker registry 更好地服务于生产环境的一些技巧。

在本章稍后还会介绍修改本地 Docker 服务使用 Docker registry 的方法，包括如何替换 Docker.io 这个默认 registry，以及如何添加其他 registy 到搜索路径中。



注意

使用 Docker 并不需要搭建本地 registry。然而，当跟随本书构建、保存和重用镜像时，你会发现有办法保存自己的镜像（尤其是私有镜像）——而不是将它们推送到公共 Docker Hub Registry 中——是很方便的。这就是说，如果想更多地了解如何使用容器，现在可以先跳过本章，在需要着手搭建 Docker registry 时再回来阅读。

3.1 获取和启动私有 Docker registry

可以通过多种不同的方式在 Linux 系统上运行 Docker registry。对于包含了 docker-registry 软件包的 Linux 发行版（如 Fedora 和 Red Hat Enterprise Linux），可以安装该软件包并启动相应的服务。至于其他发行版，可以运行来自 Docker.io 的官方 registry 容器镜像，该镜像提供了对应的 registry 服务。

本章稍后的小节会针对具体 Linux 系统进行说明，请参考所使用的 Linux 的说明，了解如何在其上安装和运行 Docker registry。我会用 Fedora 来说明如何使用 docker-registry 软件包，并用 Ubuntu 来展示如何使用 registry 容器。

下面是搭建 Docker registry 需要了解的一些常识。

- **安装在任何位置**：像多数服务器一样，Docker registry 不需要在客户系统上（也就是运行 `docker` 命令的地方）安装任何东西。可以将它安装在客户程序能够通过网络访问到的任何 Linux 系统中。那样一来，多个 Docker 客户程序都可以访问 Docker register。
- **开放端口**：如果 Docker registry 没有安装在 Docker 客户程序所在的机器上，那就必须确保在 Docker registry 运行的机器上防火墙没有阻塞 TCP 的 5000 端口。
- **提供空间**：如果要推送大量镜像到自己的 registry 中，磁盘空间会很快耗尽。如果使用 docker-registry 软件包，镜像文件会保存在`/var/lib/docker-registry` 目录中。因此，要确保为该目录配置足够的磁盘空间，或者配置一个不同的目录——如果你想的话。

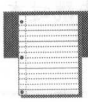
3.1.1 在 Fedora 中搭建 Docker registry

根据下面的说明在 Fedora 系统中安装和启动 Docker registry。此时，该过程使用了 docker-registry 的 RPM 包创建版本 1 的 Docker registry。尽管该过程只在 Fedora 上测试过，但相同的基本过程也适用于下面这些 Linux 发行版：

- Fedora 22 或更高的版本；
- Red Hat Enterprise Linux 7.1 或更高的版本；

- CentOS 7.1 或更高的版本。

Atomic 项目对应的 Fedora、RHEL 和 CentOS 发行版不包含 docker-registry 软件包，所以在 Atomic Linux 系统中搭建 registry 就只能使用 registry 容器（这会在之后介绍在 Ubuntu 中搭建 Docker registry 时涉及）来获得该特性。



注意

在下面的过程中，将使用镜像标签（image tag）来确定 registry——那是你打算保存镜像的位置。要更深入地了解标签，参见第 6 章。为了使 docker-registry 工作，需要编辑 usr/lib/system/docker-registry.service 文件，去掉--debug 选项。

（1）**安装 docker-registry**。在 Fedora 中安装 docker-registry 软件包时，会同时安装其依赖的数十个软件包。输入下面的命令来安装这些软件包：

```
# yum install docker-registry
...
Transaction Summary

================================================
Install 1 Package (+15 Dependent packages)
Total download size: 6.8 M
Installed size: 39 M
Is this ok [y/d/N]: y
```

（2）**列出 docker-registry 的内容**。使用 rpm 命令列出 docker-registry 的内容，这个软件包中有近 200 个文件（大多数是 Python 代码）。下面的命令只展示文档和配置文件（之后我会介绍如何配置它们）：

```
# rpm -ql docker-registry | grep -E "(/etc)|(/usr/share)|(systemd)"
/etc/docker-registry.yml
/etc/sysconfig/docker-registry
/usr/lib/systemd/system/docker-registry.service
/usr/share/doc/docker-registry
/usr/share/doc/docker-registry/AUTHORS
/usr/share/doc/docker-registry/CHANGELOG.md
/usr/share/doc/docker-registry/LICENSE
/usr/share/doc/docker-registry/README.md
```

（3）**打开防火墙**。如果 Fedora 系统启用了阻塞进入连接的防火墙，可能需要打开 TCP 的 5000 端口，以允许访问 Docker registry 服务。这里假设你使用的是 Fedora 的防火墙服务，运行下面的命令来打开防火墙的这个端口（这是立即并永久生效的），并查看该端口是否已经打开：

```
# firewall-cmd --zone=public --add-port=5000/tcp
# firewall-cmd --zone=public --add-port=5000/tcp --permanent
```

```
# firewall-cmd --zone=public --list-ports
5000/tcp
```

（4）**启动 docker-registry 服务**。如果要对 Docker registry 进行特殊配置，在启动服务前先参考接下来的几节，但对于简单的 docker-registry 安装，只要按如下方式启动服务就可以开始使用了（按其状态显示，docker-registry 服务已启用并处于活动状态）：

```
# systemctl start docker-registry
# systemctl enable docker-registry
Created symlink from
  /etc/systemd/system/multi-user.target.wants/docker-registry.service
  to /usr/lib/systemd/system/docker-registry.service.
# systemctl status docker-registry
docker-registry.service - Registry server for Docker
 Loaded: loaded (/usr/lib/systemd/system/docker-registry.service;enabled)
 Active: active (running) since Mon 2015-05-25 12:02:14 EDT; 42s ago
Main PID: 5728 (gunicorn)
  CGroup: /system.slice/docker-registry.service
      ├─5728 /usr/bin/python /usr/bin/gunicorn --access-logfile
        - --max-requests 100 --graceful-timeout 3600-t 36...
...
```

（5）**获取镜像**。hello-world 镜像是 Docker Hub Registry 提供的一个用于测试 Docker 的常用镜像。按如下方式运行该镜像（这会将该镜像拉取到本地系统并运行它）：

```
# docker run --name myhello hello-world
Unable to find image'hello-world:latest' locally
latest: Pulling from docker.io/hello-world
91c95931e552: Download complete
a8219747be10: Download complete
Hello from Docker.
docker.io/hello-world:latest: The image you are pulling has been
verified.
...
```

（6）**允许访问 registry**。Fedora 和 Red Hat Enterprise Linux 的 docker 客户端要求从 registry 那里获得证书，或者将 registry 标记为不安全的。就本例而言，可以通过编辑 /etc/sysconfig/docker 文件将 registry 标记为不安全的，具体请在文件中插入下面的内容：

```
ADD_REGISTRY='--add-registry localhost:5000'
INSECURE_REGISTRY='--insecure-registry localhost:5000'
```

编辑完成后，重启本地 Docker 服务：

```
# systemctl restart docker
```

（7）**为镜像打标签**。使用 docker tag 为镜像指定一个名字，之后可以用这个名字把镜像推送到本地系统的 Docker registry 中：

```
# docker tag hello-world localhost:5000/hello-me:latest
```

（8）**推送镜像**。输入下面的命令，将 hello-world 推到本地 Docker registry 中：

```
# docker push localhost:5000/hello-me:latest
The push refers to a repository [localhost:5000/hello-me] (1 tags)
...
Pushing tag for rev [91c95931e552] on
     {http://localhost:5000/v1/repositories/hello-me/tags/latest}
```

（9）**拉取镜像**。为了确保能够从 registry 中获取镜像，删掉本地镜像，再打开一个终端，然后尝试从本地 registry 取回该镜像：

```
# docker rm myhello
# docker rmi hello-world localhost:5000/hello-me:latest
# docker pull localhost:5000/hello-me:latest
Pulling repository localhost:5000/hello-me
91c95931e552: Download complete
a8219747be10: Download complete
# docker images
REPOSITORY                  TAG      IMAGE ID       CREATED      VIRTUAL SIZE
localhost:5000/hello-me     latest   91c95931e552   5 weeks ago  910 B
```

如这个例子所示，已经将镜像成功地推送到本地 registry 中，并从中成功地拉取回来。此时，有两个选择：

- 如果想要更多地了解 Docker registry 如何工作以及如何修改其行为，可以跳到本章后面的 3.2 节；
- 如果准备好开始使用 Docker 容器，可以直接跳到第 4 章。

接下来介绍如何在 Ubunbu 中搭建 Docker registry。

3.1.2 在 Ubuntu 中搭建 Docker registry

除了通过软件包来安装 Docker registry，还可以从 Docker Hub Registry 下载 registry 容器，并使用该容器来提供 Docker registry 服务。这是简单快速地尝试 Docker registry 的一种方式，但这种默认的 registry 在规模上不能很好地扩展以用于生产环境，而且也非常难配置。



注意

有若干版本的 registry 可供使用。这个示例使用了 registry:latest，这会生成版本 1 的 Docker registry 的镜像。到你尝试的时候，可能已经有稳定的版本 2 可以使用了。建议参考 https://docs.docker.com/registry/，了解如何运行版本 2 的 Docker registry。

虽然下面的过程只在 Ubuntu 14.04 上测试过，但相同的基本过程应该适用于任何运行 Docker 服务的 Linux 系统。

准备开始前，请先按第 2 章的介绍安装 Docker，并启动 Docker 服务。我建议打开两个

终端窗口（shell）来完成下面的过程。打开其中一个终端窗口运行 registry 服务，以便能够在启动和测试 registry 时观察其处理过程，打开另一个终端窗口可以推送和拉取镜像。

（1）**获取 registry 镜像**。按如下方式执行 `docker pull` 命令，从 Docker Hub Registry 中拉取 registry 镜像（参见第 5 章中对 `docker pull` 命令的描述）：

```
$ sudo docker pull registry:latest
Pulling repository registry
204704ce3137: Download complete
e9e06b06e14c: Download complete
...
```

（2）**运行 regsitry 镜像**。要尝试 Docker registry 时，请先在前台运行镜像，这样就能够观察容器镜像运行中产生的消息（参见第 4 章中对 `docker run` 的描述）。下面的命令会启动最新的 registry 镜像，暴露系统的 TCP 5000 端口以便容器外部的客户能够使用它，谨记在第一个终端中以前台进程来执行该命令：

```
$ sudo docker run -p 5000:5000 registry:latest
[2015-05-25 21:33:35 +0000][1][INFO] Starting gunicorn 19.1.1
[2015-05-25 21:33:35 +0000][1][INFO] Listening at: http://0.0.0.0:5000 (1)
[2015-05-25 21:33:35 +0000][1][INFO] Using worker: gevent
...
```

（3）**获取镜像**。打开第二个终端窗口，测试能否推送和拉取镜像。`hello-world` 镜像是 Docker Hub Registry 提供的一个用于测试 Docker 的常用镜像。按如下方式运行该镜像（这会将该镜像拉取到本地系统并运行它）：

```
$ sudo docker run --name myhello hello-world
Pulling repository hello-world
91c95931e552: Download complete
a8219747be10: Download complete
Hello from Docker.
This message shows that your installation appears to be working
correctly.
 ...
```

（4）**为镜像打标签**。使用 `docker tag` 为镜像指定一个名字，之后将用这个名字把镜像推送到本地系统的 Docker registry 中：

```
$ sudo docker tag hello-world localhost:5000/hello-me:latest
```

（5）**推送镜像**。输入下面的命令，将 `hello-world` 推到本地 Docker registry 中：

```
$ sudo docker push localhost:5000/hello-me:latest
The push refers to a repository [localhost:5000/hello-me] (len: 1)
...
Pushing tag for rev [91c95931e552] on
    {http://localhost:5000/v1/repositories/hello-me/tags/latest}
```

（6）**检查 Docker registry 的日志消息**。如果镜像成功推送到 registry 中，第一个终端会显示 PUT 命令成功的消息。例如：

```
172.17.42.1 - - [25/May/2015:22:12:37 +0000] "PUT
/v1/repositories/hello-me/images HTTP/1.1" 204 - "-" "docker/1.0.1
go/go1.2.1 git-commit/990021a kernel/3.13.0-24-generic os/linux
arch/amd64"
```

（7）**拉取镜像**。为了确保能够从 registry 中获取镜像，在第二个终端中删掉本地镜像，然后尝试从本地 registry 取回该镜像：

```
$ sudo docker rm myhello
$ sudo docker rmi hello-world localhost:5000/hello-me:latest
$ sudo docker pull localhost:5000/hello-me:latest
Pulling repository localhost:5000/hello-me
91c95931e552: Download complete
a8219747be10: Download complete
# docker images
REPOSITORY                TAG     IMAGE ID      CREATED      VIRTUAL SIZE
localhost:5000/hello-me latest 91c95931e552 5 weeks ago 910 B
```

（8）**再次运行 docker registry**。这次不再以前台运行 registry 的镜像，保持终端窗口打开，我们让它在后台（-d）更为持久地运行。为此，先停止正在运行的 registry 容器，然后按如下方式启动新镜像：

```
$ sudo docker run -d -p 5000:5000 registry:latest
```

Docker registry 现在已在后台运行，并准备好进行使用。此时，有两个选择：

- 如果想要更多地了解 Docker registry 如何工作以及如何修改其行为，可以跳到本章稍后的 3.2 节；
- 如果准备好开始使用 Docker 容器，可以直接跳到第 4 章。

接下来介绍如何配置 Docker registry。

3.2 配置私有 Docker registry

docker-registry 包或者 registry 容器中提供的默认 registry 很适合用来尝试 Docker registry。但是，如果打算在生产环境中使用 registry，则需要更深入地了解如何配置 Docker registry，使其更好地满足需求。

接下来几节介绍如何调整 Docker registry，既会涉及 docker-register 包的情况，也会涉及 registry 容器的使用。

3.2.1 配置 docker-registry 包

为了更好地理解 docker-registry 包的工作方式，我们先来介绍默认情况下 registry 是如何

开始运行的。当 Fedora 或 Red Hat Enterprise Linux 中的 docker-registry 服务启动时，它会启动 gunicorn 进程。默认会运行 1 个 gunicorn 主进程和 4 个 gunicorn 工作进程（worker）来提供该服务。

由 gunicorn 进程的 ps 输出可以看出其设置情况：

```
# ps -ef | grep gunicorn
00:00:00 /usr/bin/python /usr/bin/gunicorn --access-logfile -
 --max-requests 100 --graceful-timeout 3600 -t 3600 -k gevent -b
0.0.0.0:5000 -w 4 docker_registry.wsgi:application
```

下面列出了可以从这行命令输出了解到的配置信息。

- **--access-logfile**：对 docker-registry 服务的访问会记录到所设置的任何文件中。对于本例，日志文件被设置为一个连字符（-），因此访问信息会直接发送给标准输出（这种情况下，它们可以被 systemd 日志拾取，并可以用 journalctl 命令来查看）。
- **--max-requests 100**：设置 gunicorn 守护进程能够接收的请求上限为 100。达到上限后，工作进程会重启。
- **--graceful-timeout 3600**：向 gunicorn 工作进程发送重启信号后，再给它 3600 秒（6 分钟）来完成正在处理的请求。如果它不能在这段时间内完成请求处理，它会被杀死（kill）。
- **-t 3600**：如果 gunicorn 工作进程超过 3600 秒（6 分钟）没有活动，它会被杀死并重启。
- **-k gevent**：将 gunicorn 工作进程的类型设置为 gevent（基于 Greenlets 的一种异步工作进程）。
- **-b 0.0.0.0:5000**：将工作进程绑定到系统的所有 IP 地址（0.0.0.0）的 5000 端口上。这让 docker 客户端可以使用 TCP 5000 端口通过系统的任意外部网络接口连接到 Docker registry。
- **-w 4**：设置工作进程的数量为 4（要高于初始 gunicorn 进程的数量）。
- **docker_registry.wsgi:application**：进程随 Docker registry wsgi 应用程序一起运行。

可以通过修改/etc/sysconfig/docker-registry 文件来改变 docker-registry 服务的行为。在 Fedora 中，该文件的默认设置如下：

```
# The Docker registry configuration file
DOCKER_REGISTRY_CONFIG=/etc/docker-registry.yml

# The configuration to use from DOCKER_REGISTRY_CONFIG file
SETTINGS_FLAVOR=local

# Address to bind the registry to
REGISTRY_ADDRESS=0.0.0.0
```

```
# Port to bind the registry to
REGISTRY_PORT=5000

# Number of workers to handle the connections
GUNICORN_WORKERS=4
```

在 docker-registry 文件中，可以让 Docker registry 只监听一个特定的 IP 地址（默认情况下，REGISTRY_ADDRESS=0.0.0.0 会监听所有 IP 地址），可以将服务的端口改成非 TCP 的 5000 端口，或者将 gunicorn 工作进程的数量改成其他数量（而不是 4）。

/etc/docker-registry.yml 文件被设置为 Docker registry 的配置文件。SETTINGS_FLAVOR=local 会让配置文件包含公共变量，并将目录/var/lib/docker-registry 用于本地存储。可以在/etc/sysconfig/docker-registry 文件中设置的公共变量如下。

- LOGLEVEL：默认情况下，日志级别被设置为 info。日记级别可以被设置为 debug、notice、warnning、warn、err、error、crit、alert、emerg 或 panic。
- DEBUG：将其设置为 true 或 false 来打开或关闭调试。
- STANDALONE：如果设置为 true（默认），registry 将作为独立的 registry，不会查询 Docker 索引（index）。
- INDEX_ENDPOINT：如果本地 registry 没有按默认设置为独立运行（standalone），索引端点（index endpoint）将设置为 https://index.docker.io。
- STORAGE_REDIRECT：默认是 disabled。
- DISABLE_TOKEN_AUTH：如果服务不是独立运行的，该变量要设置为 enabled，以便允许令牌验证（token authentication）。
- PRIVILEGED_KEY 默认：不设置。
- SEARCH_BACKEND 默认：没有搜索后端。
- SQLALCHEMY_INDEX_DATABASE：SQLite 搜索后端数据库（SQLite search backend database）默认会设置为 sqlite:////tmp/docker-registry.db。

如果不想使用本地的 setting flavor，可以看看/etc/docker- registry.yml 文件。不同的 setting flavor 可以用于 Ceph Object Gateway 配置、Google Cloud Storage 配置、OpenStack Swift Storage 配置以及其他情况。

还可以设置 gunicorn 进程使用的其他变量，这些变量如下。注意，其中一些值在之前 gunicorn 命令行中出现过。

- GUNICORN_GRACEFUL_TIMEOUT：设置工作进程平滑重启的超时时间（以秒为单位）。
- GUNICORN_SILENT_TIMEOUT：设置重启已失效的工作进程的超时时间（以秒为单位）。
- GUNICORN_USER：如果不以 root 用户权限来运行，在这里设置运行 gunicorn 进程的用户。

- GUNICORN_GROUP：如果不以 root 用户组权限来运行，在这里设置运行 gunicorn 进程的用户组。
- GUNICORN_ACCESS_LOG_FILE：设置记录服务访问消息的日志文件名。默认情况下，消息会通过标准输出传送到 systemd 日志。
- GUNICORN_ERROR_LOG_FILE：设置记录出错消息的日志文件名。默认情况下，消息会通过标准输出传送到 systemd 日志。
- GUNICORN_OPTS：确定要传给 gunicorn 进程的额外选项。

在设置或修改了`/etc/sysconfig/docker-registry`文件的变量后，重启`docker-registry`服务以使其生效。

3.2.2 配置 `registry` 容器

当尝试配置 registry 容器镜像时，容器镜像的创建者建议重新构建 registry 容器镜像，而不是修改正在运行的容器的内容。特别是，你可能想为自己的 registry 添加安全度量以及更灵活的存储特性。

到目前为止，本书尚未介绍那些用来构建你自己的容器的概念。不过，在熟悉整个过程后，如果你打算构建一个版本 1 的自定义 regsitry 容器，我建议参考`docker-registry`的 GibHub 主页：https://github.com/docker/docker-registry。

在`docker-registry`的 GitHub 主页上，可以找到构建版本 1 的 registry 镜像的信息以及用来构建该镜像的 Dockerfile 的链接（https://github.com/docker/docker-registry/blob/master/Dockerfile）。

在读者阅读本书时，Docker registry 的版本 2 也许已经可以使用了。请访问 Docker registry 2.0 的主页（https://docs.docker.com/registry），以了解如何部署和配置新版本的 Docker registry。

3.3 了解 Docker 镜像命名空间

与互联网使用域名系统（Domain Name System，DNS）的方式（用唯一的一组名字来代表世界上所有的主机）相似，Docker 也着手创建了一个命名空间，这个命名空间允许用唯一的方法来命名全世界每一个容器镜像。在这种情况下，`docker run` *`someimage`* 会把名字完全相同的 *`someimage`* 拉取到本地系统并运行，而不管你身在何处以及执行该命令的 Linux 系统是何种类型。

对某些 Docker 的潜在用户来说，这带来了一些问题。有些 Docker 安装是无法访问互联网的，还有些安全要求只允许从自己的 registry 中搜索和拉取镜像。这些问题使得在上述环境中无法安装一个纯粹的 Docker 系统。

Docker 镜像命名空间的工作方式一直承受着要求改变其某些方面的压力，因此你大可以期望情况会随着时间而改善。但在目前情况下，你应该了解纯粹基于上游 Docker Project 代码运行的 Docker 系统具有以下特点。

- **搜索**：运行 `docker search` 命令时，目前没有打过补丁的 Docker 系统只会搜索 Docker Hub Registry。
- **阻塞 registry**：Docker 没有阻塞 Docker Hub Registry 的功能。因此，没有明确给出具体 registry 的情况下拉取镜像，会导致 Docker 在 Docker Hub Registry 上搜索该镜像（如果本地系统中还没有该镜像）。
- **改变默认 registry**：Docker 不能将默认 registry 从 Docker Hub Registry 变为其他 registry。
- **推送确认**：在开始推送镜像前，Docker 不会询问你以获得确认。

Docker 社区一直在讨论对这些特性进行调整。改变这些特性的工作方式的补丁已经包含在 Red Hat Enterprise Linux、Fedora、Atomic project 以及相关的 Linux 发行版中。例如，RHEL Atomic 的 docker 软件包的当前版本（docker-1.8）就包含刚刚提到的一些特性。

下面的设置来自 RHEL Atomic 系统的`/etc/sysconfig/docker` 文件，它们代表了尚未加入到上游 Docker Project 的特性：

```
ADD_REGISTRY='--add-registry registry.access.redhat.com'
# BLOCK_REGISTRY='--block-registry'
# INSECURE_REGISTRY='--insecure-registry'
```

ADD_REGISTRY 变量可以让用户添加 `docker search` 和 `docker pull` 命令所使用的 registry。对于 Red Hat 发行版的用户，这会将 Red Hat 自己的 registry（`registry.access.redhat.com`）放到 Docker Hub Registry 的前面，因此 Red Hat 的用户知道自己会先从该 registry 搜索和拉取镜像。用户也可以将该 registry 替换为自己的 registry 或将自己的 registry 放在 Red Hat 的 registry 之前。

使用该配置文件的 ADD_REGISTRY 变量将用户添加的 registry 置于搜索列表的前面。然后，如果在用户添加的所有 registry 中都没有找到所请求的镜像，那么接下来仍会搜索 Docker Hub Registry。想要改变这个行为，需要使用 BLOCK_REGISTRY 变量。

通过设置 BLOCK_REGISTY 变量，用户可以阻塞对自己选定的任何 registry 的访问。当然，目前默认搜索的只有 Docker Hub Registry。因此，要想阻塞对 Docker Hub Registry 的搜索和拉取请求，可以使用下面的配置：

```
BLOCK_REGISTRY='--block-registry docker.io'
```

设置后，即使这些镜像存在于 Docker Hub Registry 中，任何镜像请求如果不能在 ADD_REGISTRY 变量指定的 registry 集合中找到就会报告查找失败。如此一来，这个特定 docker 安装的用户就只能从你具体指定的 registry 中搜索镜像。

INSECURE_REGISTRY='--insecure-registry'并不是显式地允许或禁止一个 registry。这是用于处理一种特殊情况：某人想使用本地 Docker 客户端从一个使用 HTTPS 进行通信的 registry 拉取镜像，但该客户端没有 registry 提供的证书来验证其真实性。去掉该变量所在行的注释并添加非安全 registry 可以容许 `docker` 命令在完全没有授权的情况下从对应 registry

中拉取镜像。例如：

```
INSECURE_REGISTRY='--insecure-registry myreg.example.com'
```

再次提醒，刚刚描述的这个特性以及其他特性都不属于上游的 Docker Project。但是，如果你的安装需要这些特性，可以改变在 Docker 中访问 registry 的默认方式，目前这些特性可以在 Fedora、RHEL、CentOS 和相关的 Atomic 项目系统中使用。

3.4 小结

搭建私有 Docker registry 让用户具备不使用公有 Docker Hub Registry 就能拉取和推送镜像的能力。本章介绍了两种不同的搭建 Docker registry 的方法。

对于包含 docker-registry 包的 Linux 发行版（如 Fedora 和 Red Hat Enterprise Linux），可以安装该软件包并通过 `systemctl` 命令启动 docker-registry 服务。另外一个选择是，任何运行 Docker 服务的系统都可以拉取和运行 registry 镜像来提供私有 Docker registry，这个镜像可以从 Docker Hub Registry 中获取。

除了介绍如何搭建自己的 Docker registry，本章还描述了 Docker 镜像命名空间以 Docker Hub Registry 为中心的工作方式。Fedora 和其他 Red Hat 发起的操作系统已经实现了对该模型的建议修改，而 Docker 社区正在讨论这些建议修改。这些修改让用户能够指定 Docker 服务的搜索和拉取请求将使用哪个 regsitry。

第二部分

玩转单个容器

第 4 章

运行容器镜像

本章主要内容

- 从容器运行交互式命令
- 在容器中运行管理命令
- 从容器内运行服务
- 运行特权容器

一旦在本地 Linux 系统中安装并运行了 Docker 服务，运行首个容器就非常简单了。开始时，用户的系统无需安装任何 Docker 格式的镜像。docker run 命令会到系统外为用户寻找镜像。对这个示例而言，如同我将要展示的那样，你仅需要几百兆字节的磁盘空间以及等待镜像下载的一点点时间。



注意

每次运行容器镜像或者将容器提交（commit）给镜像时，都会消耗磁盘空间。如果在运行多个容器后，开始遇到磁盘空间耗尽的错误，可以使用 docker rm 命令（删除容器）和 docker rmi 命令（删除镜像）来释放一些空间。参见第 11 章，了解更多信息。

假定你能连接互联网，你可以按如下方式运行第一个容器：

```
# docker run fedora cat /etc/os-release
Unable to find image 'fedora:latest' locally
00a0c78eeb6d: Pull complete
Status: Downloaded newer image...

NAME=Fedora
VERSION="22 (Twenty Two)"
ID=fedora
...
```

该 docker run 命令运行中发生了几件有趣的事情。例如：

- **确定要运行的镜像和命令**：docker 命令行表达的是，找到名为 fedora 的容器镜像，把它下载到本地系统中，而后在该容器中运行 cat /etc/os-release 命令（查看容器中的操作系统版本）。由于镜像名的后边没有附加标签，所以会假定是：latest。
- **寻找镜像**：Docker 首先在本地系统查找 fedora:latest 镜像。由于本地没有该镜像，Docker 就搜索 Docker.io 的 regsitry，在那里找到并下载 fedora 镜像。如果使用的是 Red Hat Enterprise Linux，docker 命令会在查找 Docker.io 的 registry（Docker Hub Registry）前，先查找 Red Hat 的 registry。
- **执行命令**：该示例中，cat 命令会在显示器上显示/etc/os-release 文件的内容。请记住，容器有自己的文件系统，因此，该输出告诉我们这个容器是由 Fedora 22 系统构建的。如果该容器是运行在 Ubuntu 系统上或是 Red Hat Enterprise Linux 系统上，其产生的结果是一样的。

至此，fedora 镜像已存在于本地系统中，下次再对该镜像执行 docker run 命令时，就不需要重新下载了。该命令会从本地存储的镜像上开始运行。

在继续尝试其他 docker run 命令前，请谨记镜像与容器之间的这些区别。

- **镜像**：这是一个永久存储的容器实例。docker images 命令会显示系统内的镜像。docker rmi*image* 命令会删除指定的镜像。只要有所帮助，你可以为同一个镜像分配多个别名（包括名字和标签）。使用 docker run 命令运行一个镜像。
- **容器**：容器是通过运行镜像创建的。如果容器在后台运行（分离模式），那它能够在 docker run 命令退出后继续运行。使用 docker ps 命令可以查看正在运行的容器。容器一旦退出（除非明确地删除它），该容器的状态会得到保存。通过 docker ps -a 命令能够查看那些已经保存且不再运行的容器。docker start *container* 命令能够再次启动一个容器（这包括镜像和对该运行实例所做的任何改变）。docker stop *container* 能够停止一个运行的容器。请将上述命令中的 *container* 替换为运行镜像时所创建容器的名字或容器 ID。

现在，运行几个命令来感觉一下刚刚运行容器镜像的结果：

```
# docker images
REPOSITORY    TAG        IMAGE ID        CREATED        VIRTUAL SIZE
fedora        latest     834629358fe2    3 months ago   241.3 MB
# docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED     STATUS   PORTS    NAMES
# docker ps -a
CONTAINER ID    IMAGE            COMMAND                  CREATED
    STATUS                          PORTS     NAMES
a068cd24ab4b    fedora:latest    cat /etc/os-release a minute ago
    Exited (0)  a minute ago                 stoic_carson
```

docker images 命令显示 fedora 镜像保存在系统上并且可以用来再次运行。由于容器在显示了 cat 命令的输出后就执行完毕并退出了，因此 docker ps 命令显示没有正在运行的容器，而 docker ps -a 命令显示该容器还可以再次启动。

由于没有给该容器指定名字，Docker 为其随机分配了一个名字（这里是 stoic_carson）。如果想要再次运行该容器（这个容器包括了镜像、添加到容器中的任何内容以及任何选项），运行 docker start 命令并提供容器的 ID 或名字：

```
# docker start -i a068cd24ab4b
...
NAME=Fedora
VERSION="22 (Twenty Two)"
ID=fedora
VERSION_ID=22
...
```

请注意，docker start 命令会取得容器退出时的状态（可能被停止了，或者如同本示例，容器已完成了它的任务）并完全重新运行之前的 docker run 命令——包括作为参数的 cat 命令。我只是添加了-i 选项，这会将容器的输出定向到本地 shell（STDOUT）。

docker run 命令有诸多选项。本章接下来的几节会继续使用 fedora 镜像来解释 docker run 命令提供的这些选项。

4.1 以交互方式运行容器镜像

当运行容器镜像时，它会执行你提供的命令（或是构建在其中的命令），然后退出。如果想让命令在后台持续运行（就如同处理持续运行的服务那样，如 Web 服务器或者打印服务器），可以分离它（用-d 选项），使其在后台执行。如果想让其在前台运行，可以采用交互式的方式运行它（-i 选项），而且通常还会打开一个终端会话（-t 选项）。

本章后面的 4.2 节会给出以分离模式运行容器镜像的示例。接下来的这一节将展示以交互方式运行容器镜像的示例。

4.1.1 启动交互 Bash shell

一个常见的交互式容器的例子是：打开一个 shell 在容器内部直接进行操作。运行一个交互式容器镜像能够让你有机会查看容器内部的情况并更改其内容。



注意

基本容器镜像（如这里使用的 fedora 镜像）的内容在本书写作过程中已经变了好几次。如果 ps 命令没有包含在最新版的 fedora 镜像中，输入 yum install procps，从容器内安装它。

在下面这个例子中，我通过执行/bin/bash 命令来打开 fedora 镜像中的 shell。-i 和-t 选项能够让我的当前终端会话与容器内的 shell 进行交互：

```
# docker run -it fedora /bin/bash
bash-4.3#
```

现在，你得到了容器内 shell 的 bash 提示符。如何判断自己是在容器内？如果容器包含了 ps 命令（如同本示例的情况），按如下方式执行 ps 命令来查看正在运行的进程：

```
bash-4.3# ps -e
  PID TTY          TIME CMD
    1 ?        00:00:00 bash
    7 ?        00:00:00 ps
```

ps -e（该命令会列出进程表中的所有进程）的输出告诉了我们一些事情。进程 ID 为 1 的进程（在正在运行的 Linux 系统中，这通常是 init 或 systemd）是 bash 命令。目前，容器中另一个正在运行的进程是 ps 命令本身。这说明，默认情况下你看不到宿主机系统的进程表（那里可能有上百个正在运行的进程）。容器有自己的进程表。

如果有兴趣了解容器内的其他东西，可以运行下面这些命令：

```
bash-4.3# rpm -qa | more                查看已安装的软件包
bash-4.3# rpm -ql curl                  查看某个已安装软件包的内容
bash-4.3# man curl                      无法查看联机帮助页面
bash: man: command not found
bash-4.3# ip addr show                  无法查看网络接口
bash: ip: command not found
```

所以，根据容器镜像创建时所包含的内容，你也许无法在容器中运行所有想要运行的命令。但是，请记住，容器意在简洁，它只包括那些它需要运行的组件。否则，你可能要使用完整的虚拟机。

但不要失望。来自主要 Linux 发行版的多数基础容器镜像都包含了向正在运行的容器添加软件的方法。基于 fedora 的镜像也不例外。假设宿主机系统能够连接互联网，并且已经配置 Docker 为每个容器提供专用的 IP 地址（这是默认配置），就可以使用容器中的 yum 或 dnf（Fedora 和 RHEL）或者 apt-get（Ubuntu 和 Debian）来添加软件。

继续使用正在运行的 fedora 镜像，可以运行 yum 命令添加更多软件到容器中。通常，应该在最初构建镜像时添加软件。但为了本章的剩余部分能够尝试 docker run 命令，运行下面的 yum 命令来添加我们之后要使用的软件：

```
bash-4.3# yum install iproute net-tools bsd-games words \
          vsftpd httpd httpd-manual -y
Resolving Dependencies
--> Running transaction check
...
Complete!
bash-4.3# exit
```

输入 exit 来结束 shell 会话并离开容器。此时，我想把该容器保存为一个镜像，这样我就能复用它来尝试本章中的其他 run 命令了。我是这么做的：

```
# docker ps -a
CONTAINER ID   IMAGE            COMMAND       CREATED
   STATUS                       PORTS  NAMES
88f6c09523b5   fedora:latest    "/bin/bash"   3 hours ago
   Exited (0) 7 seconds ago                  trusting_heisenberg
# docker commit -a "Chris Negus" 88f6c09523b5 testrun
# docker images
REPOSITORY  TAG       IMAGE ID       CREATED          VIRTUAL SIZE
fedora      latest    834629358fe2   1 month ago      422.2 MB
testrun     latest    226f7543f12a   3 minutes ago    431.6 MB
```

现在，本地系统已经有了名为 testrun 的镜像并可以用于接下来的 docker run 示例。

4.1.2 玩些基于字符的游戏

在上一节创建的 testrun 镜像中，我安装了 bsd-games 软件包。该软件包含有早期 UNIX 系统上的基于字符的游戏。这些游戏提供了一种有趣的方式来尝试从 Docker 容器运行屏幕导向的游戏。

由于想运行容器，然后让容器在游戏结束时销毁掉，我向命令行添加了--rm 选项。这是应该养成的好习惯，这样，在使用完容器后不会有太多不想要的容器占满磁盘空间。

下面展示了如何从容器内的交互终端玩 Hangman：

```
# docker run -it --rm testrun /bin/hangman
 _______
  |     |
  |     O            Guessed: abcdeimnoprstu
  |    /|\
  |     |            Word #:          2
  |    /             Current Average: 7.500
__|____              Overall Average: 9.000
|     |___
|_________|

  Word:  cambodian
Guess:
You got it!
Another word?
```

下面是启动 Snake 游戏的方法，这里要控制方向键来尝试抓取$符号，并在蛇吃掉你之前逃脱：

```
# docker run -it --rm testrun /usr/bin/snake
|-------------------------------------------------|
| $150                          $                 |
|                            I                    |
|                   ss                            |
|       #          s ss                           |
|                    S                            |
|-------------------------------------------------|
```

下面是启动 adventure 来开始探索 Colossal Cave 的方法：

```
# docker run -it --rm testrun /usr/bin/adventure
Welcome to Adventure!!  Would you like instructions?
n
You are standing at the end of a road before a small brick building.
Around you is a forest. A small stream flows out of the building and down a gully.
building
You are inside a building, a well house for a large spring.
There is a shiny brass lamp nearby.
...
take lamp
OK
xyzzy
It is now pitch dark. If you proceed you will likely fall into a pit.
light
Your lamp is now on.
quit
```

如果想继续探险，输入方向（compass directions）、up、down 或者房间名称来在洞穴中四处走动。捡起你发现的物品并把它们带回到最初的小砖房里来得分。

4.1.3 在容器内运行管理命令

为 testrun 镜像添加 ip-route 和 net-tools 软件包，以便在容器内部可以运行 ip 和 route 这样的命令。下面是在容器内运行这些命令的示例，以及这些命令让我们了解到的容器网络接口的情况：

```
# docker run -it --rm testrun /usr/sbin/ip addr show eth0
165: eth0: <NO-CARRIER,BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state DOWN group default
    link/ether 02:42:ac:11:00:24 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.36/16 scope global eth0
...
# docker run -it --rm testrun /usr/sbin/route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         172.17.42.1     0.0.0.0         UG    0      0        0 eth0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 eth0
# ip addr show docker0
5: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
group default
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
       valid_lft forever preferred_lft forever
```

上面的示例中，前两个 docker 命令让我们了解到容器内的网络情况，接下来的 ip 命令显示了宿主机的 docker0 网络接口。注意，宿主机的 docker0 接口有一个 172.17. 42.1/16

的 IP 地址。默认情况下，宿主机会在容器启动时将通过 DHCP 获得的地址分发给容器。testrun 运行的第一个 ip 命令显示其拥有 172.17.0.36/16 的 IP 地址。每次以这种方式运行 ip 命令都会看到一个新的 IP 地址，因为每次镜像会启动一个新的容器。

注意，route 命令显示默认网关是 172.17.42.1。通过宿主机上运行的 ip 命令，可看到 172.17.42.1 是宿主机上 docker0 接口的 IP 地址。通过宿主机的 docker0 接口进行路由能够让容器访问本地宿主机之外的网络资源。

可以使用 yum 或 apt-get 命令为容器添加需要的包，从而扩展可以在容器内部运行的命令。在第 7 章中，我会添加更多命令并运行专门的 docker 选项来进一步探索容器内部。

能够以交互方式从容器内运行命令固然是件好事，但容器真正的价值来自对提供了一个或多个服务的更为复杂的应用程序进行容器化。当容器化持久服务时，如 Web 服务器或文件服务器，docker run 使用了许多更有趣的选项。接下来介绍如何使用 docker run 来运行服务。

4.2 运行容器化服务

与直接在宿主机上运行服务相比，从容器内运行服务有更多优势。其中一些优势如下。

- **配置**：将服务放入容器，能够提前配置好提供服务所需的全部可执行程序、库、配置文件和其他元素，如此一来就无需担心宿主机系统是否提供这些组件。而且，如果需要，还很容易将容器迁移到另一个宿主机上。
- **隔离**：因为每个容器都有自己单独的文件系统和网络接口，所以能够按需要运行多个相同的服务容器。理论上，一个宿主机能够运行 100 个独立的 Web 服务器容器，只要每个容器使用各自的 IP 地址和端口来公开其服务，这些服务之间就不会彼此冲突。

docker run 用来启动独立服务容器的选项做了这些事：告诉容器执行的命令在后台运行（分离模式），连接到本地宿主机系统的存储卷，开放宿主机端口使外部世界可以访问这些服务。

为了说明这些用于运行服务的 docker run 选项，接下来将展示如何从容器内运行基本的 Web 服务器。本示例是通过在 Fedora Linux 发行版上运行 fedora 容器来完成的。



注意

虽然理论上 Docker 容器可以在任何地方运行，但你会发现不是每个容器在任何运行 Docker 的系统上都能完全相同地运行。另外，需要了解的是，这个例子运行的 fedora 容器在你尝试运行的时候可能会有所不同。为了尽可能获得相同的结果，应该使用相同版本的 Linux 发行版和 Docker（至少是 Fedora 22 上的 Docker 1.7），以及相同的容器（fedora:22）。

4.2.1 运行容器化 Web 服务器

Apache Web 服务器是世界上最流行的 Web 服务器。默认情况下，Apache 会服务/var/www 目录下的内容并监听默认的 TCP 80 端口（HTTP）和 443 端口（HTTPS）。本示例中，我想

让 Apache Web 服务器服务宿主机的/var/www/html 目录下的内容。因此，我在宿主机中创建了该目录并设置了合适的 SELinux 上下文（Fedora 或 RHEL 系统需要的）。之后，我添加了一个 index.html 文件，并在其中加入了一些要显示的内容：

```
# mkdir -p /var/www/html
# chcon -R -t httpd_sys_content_t /var/www/
# echo "Is the Web Server running: YES" > /var/www/html/index.html
```

使用本章之前构建的 testrun 镜像，这里给出的是从该镜像运行 httpd 服务的一个 docker run 命令的例子（运行命令前，请确保宿主机系统上没有 httpd 服务或其他服务正在监听 80 端口或 443 端口）。

```
# docker run -d -p 80:80 -p 443:443 --name=MyWebServer \
    -v /var/www/:/var/www testrun \
    /usr/sbin/httpd -DFOREGROUND
```

让我们将该示例所用的选项分解一下（见表 4-1）。

表 4-1 docker run 示例使用的选项

选项	描述
-d	**分离模式**：告诉容器化命令在后台运行容器
-p 80:80 -p 443:443	**发布端口**：将容器端口发布到宿主机端口。冒号左边的数字是宿主机端口，右边的是容器端口。这里我们将容器的 TCP 80 端口（HTTP）和 443 端口（HTTPS）公开给宿主机的相同端口
--name=MyWebServer	**容器名**：如果打算将容器保持一段时间，为容器分配一个名字是一个好主意。之后，可以使用这个容器名来停止和启动该容器，而不再用容器 ID
-v /var/www/:/var/www/	**挂载卷**：这个选项将宿主机的目录（冒号左边）挂载到容器的目录（冒号的右边）。这里，我使用-v 共享 Apache 使用的默认目录来保存共享的 Web 内容（/var/www）
testrun	镜像：本示例中，镜像名是 testrun，我们从 fedora 的基础镜像创建了该镜像，并添加了 httpd 和其他软件包
/usr/sbin/httpd -DFOREGROUND	**令**：这里，我使用-DFOREGROUND 选项运行了 httpd 守护进程

与本章之前运行的命令不同，之前我们只是运行了命令然后退出，这个 Web 服务器示例会在后台（-d）持续运行并开始从宿主机上获取资源。按照如下方式验证 MyWebServer 容器正在运行：

```
# docker ps
CONTAINER ID      IMAGE          COMMAND                  CREATED
  STATUS      PORTS                                          NAMES
```

```
a8019984ae79        testrun:latest "/usr/sbin/httpd  -DF 1 hour ago
  Up 1 hour  0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp     MyWebServer
```

由输出可以看出，容器被分配了容器 ID（a8019984ae79），镜像的名字是 testrun: latest，运行的命令是 httpd，容器是 1 小时前运行的，并且已运行了 1 小时，宿主机上所有 IPv4 网络接口（0.0.0.0）会将对 TCP 80 端口和 443 端口的请求转发到容器内相应的端口上。

最后的信息显示，容器的名字是 MyWebServer。我们可以用这个名字来启动、停止或删除容器，还可以用这个名字在正在运行的容器中执行别的命令，如 bash shell，来看看容器的内部情况（完成后，输入 exit 退回到宿主机）。例如：

```
# docker exec -it MyWebServer /bin/bash
bash-4.3# exit
```

为了检验容器能够共享之前添加到宿主机上的/var/www/html/index.html 文件中的文本，我使用 curl 命令从宿主机上获取该文件的内容：

```
# curl http://localhost/index.html
Is the Web Server running: YES
```

因为容器的 TCP 80 端口公开给 localhost 的相同端口，应该会显示那个成功消息。

你也许还记得我早前安装了 httpd-manual 的用户手册软件包（其有内容存储在容器内的/usr/share/httpd/manual 目录中）。你可以打开本地宿主机的 Web 浏览器并输入 http://localhost/ manual 来查看这些内容。凭借这个目录以及挂载的/var/www/html 目录，httpd 服务能够共享来自容器内外的混合数据。结果如图 4-1 所示。



图 4-1 显示 httpd 容器内部提供的 Apache 用户手册（httpd-manual 软件包）

4.2.2 在容器中运行服务时进行资源限制

默认情况下，在容器运行的时候，容器在访问内存或多个 CPU 方面并没有特别限制。当使用--memory、--memory-swap、--cpu-shares 和--cpuset-cpus 选项运行容器时，

可以设定具体的内存限制或者请求一定数量的CPU。下面的例子说明了在执行docker run命令时，如何限制内存和CPU配额。

```
# docker run -d -p 80:80 -p 443:443 --name=LimitedWebServer \
    -v /var/www/:/var/www --memory=10m --memory-swap=-1 \
    --cpu-shares=256 testrun /usr/sbin/httpd -DFOREGROUND
```

使用--memory=10m限制容器能够使用的RAM为10 MB，而--memory-swap=-1说明对能够使用的交换空间的大小没有限制（需要操作系统已配置了交换空间）。下面是如何限制容器可用内存数量的另一个例子。

--memory=10m	容器可以使用10 MB的RAM。如果没有设置memory-swap，容器也可以使用2倍memory的交换空间（20 MB）
--memory=10m --memory-swap=30m	容器可以使用10 MB的RAM。交换空间和内存的总和限制在30 MB

为了限制容器对CPU的访问，可以使用--cpu-shares和--cpuset-cpus选项。如果没有设置--cpu-shares，所有容器都有相同的CPU配额。可以在1024范围内设置--cpu-shares。默认设置是--cpu-shares=0，这让容器完全占用1024个配额。前一个例子（--cpu-shares=256）表明只有1024个可用配额的25%会分配给这个容器，而系统中的其他容器将使用100%的可用配额。

--cpuset-cpus选项限制分配这台计算机上的哪个CPU核--cpuset-cpus的值可以是具体的CPU核、逗号分隔的CPU核列表，也可以是CPU核的范围。例如：

--cpuset-cpus=0,1	使用CPU核0或1
--cpuset-cpus=3	使用第4个CPU核
--cpuset-cpus=1-3	使用CPU核1、2或3

4.3 运行特权容器

多数情况下，当使用docker run运行容器时，你会想限制容器对其宿主机系统的访问。同样，你也想阻止本地系统上的其他容器访问你正在运行的容器。然而，有些情况下，你想让容器拥有更大的宿主机系统访问权限。这些容器被称为特权容器或超级特权容器。

特权容器背后的思想是，有时候容器就是专门为操作宿主机系统本身而设计的。如果没有开放权限，容器就不能访问宿主机的命名空间（如进程表、IPC命名空间和D-bus接口）。

第13章中会介绍特权容器可以访问的宿主机命名空间，以及开放这些命名空间的原因。第13章还会包括超级特权容器的例子。但此刻，我们还是先把主题放在docker run命令上。表4-2给出了一些选项，docker run命令可以用这些选项将宿主机权限开放给容器。

表 4-2　宿主机权限选项

选项	描述
`--ipc`	IPC：开放访问宿主机的进程间通信设施。默认情况下，每个容器都有私有的 IPC 功能设施
`--net=host`	**网络接口**：向容器开放直接访问宿主机的网络接口。默认情况下，每个容器都有自己的网络接口
`--pid=host`	**进程表**：容许容器访问宿主机的进程表。默认情况下，每个容器都维护有自己的进程表
`-e HOST=/host`	**宿主机文件系统**：如果设置，HOST 环境变量将告诉容器将宿主机的 root 文件系统挂载到容器上的哪个具体目录下。推荐位置是`/host`
`--privileged`	这一选项会关闭容器运行命令的安全隔离。因此，以 root 用户运行的进程应该拥有与在系统上以 root 运行任何进程时一样的权限

再次提醒，这些选项实际应用的例子参见第 13 章。

4.4　小结

`docker run` 命令用来将以 Docker 格式保存的容器镜像转换成运行的容器实例。`docker run` 命令可以利用许多选项。例如，可以使用交互式（`-i`）和终端（`-t`）选项来运行容器，以便能够通过 shell 与容器进行交互。

尽管有些容器需要交互式运行，但更常见的情况是，容器会在后台运行来为用户提供持续的服务。`docker run` 的`-d` 选项可以分离容器进程，从而使其在后台运行（通常是作为服务的守护进程）。可以公开 TCP 端口，这些端口提供了从容器到宿主机的服务，用户访问宿主机就能获得对容器的服务的访问。

既然了解了如何运行容器，下一章将帮你了解如何处理容器镜像，包括了解如何拉取、保存以及加载容器镜像。

第 5 章

查找、拉取、保存和装载容器镜像

本章主要内容

- 使用 `docker search` 查找镜像
- 从 registry 拉取镜像
- 保存和装载镜像

用户并不需要特别了解 Docker 就可以开始运行 Docker 格式的镜像。如前一章所述，只要知道镜像的名称，一个简单的 `docker run` 命令就可以下载并运行所选择的镜像。既然已经尝试了一些 `docker run` 命令，那么现在就可以深入地了解如何管理镜像了。

本章将介绍如何使用多种不同的方法来管理 Docker 格式的镜像。使用 `docker search` 可以从 Docker Hub Registry（docker.io）以及其他可能的位置找到可用的镜像。使用 `docker pull` 可以选择具体的镜像并将其下载到自己的系统中。使用 `docker save` 可以将本地系统的镜像保存为 tar 文件，而后再使用 `docker load` 命令将这个 tar 文件添加到另一个系统中。

在搜索、保存和装载镜像时，会遇到“registry”和“repository”这两个词。registry 是一个位置——在那里能够找到与许多 repository 相关联的镜像，而 repository 是一个名字——Docker 用它来代表多个镜像。例如，`docker.io/ubuntu` 这个名字代表了 docker.io registry 中的一个 repository。在这个 repository 中可能有多个镜像，它们代表不同版本的 Ubuntu 的容器。一般来说，像 `docker pull ubuntu` 这样的操作只会从 registry 拉取一个镜像到用户的系统（`docker.io/ubuntu:latest`）。然而，如果运行 `docker pull -a ubuntu`，就会获取所有与 `docker.io/ubuntu` repository 关联的镜像。

5.1 搜索镜像

Docker 项目设立了 Docker Hub Registry（https://registry.hub.docker.com/），将其作为数以千计的 Docker 格式的容器的集中访问点。从运行 Docker 服务的系统上，可以使用 `docker search` 命令搜索 Docker Hub，来寻找任何公开发布的镜像。如果愿意，可以使用自己的 Docker Hub 账户来保存自己的镜像，以及使它们成为私有镜像。

当组织位于防火墙之后，或者组织的安全策略要求他们将自己的镜像保存在自己的办公场所内时，有些 Docker 实现允许用户配置自己的 registry 或者阻止用户访问 Docker Hub。Red Hat Enterprise Linux 实现的 Docker 服务正是这样的例子：在那里，可以选择搜索和拉取镜像所使用的 registry。

接下来的一节会介绍如何通过 `docker` 命令行以及 Web 浏览器来搜索 Docker Hub Registry。之后，读者会了解到如何设置系统，让 `docker` 命令使用 Docker Hub 之外的 registry 搜索和拉取镜像。

5.1.1 用 docker 命令搜索镜像

多数 Docker 服务的实现被配置成从 Docker Hub 查找镜像。在 Ubuntu、Fedora 和其他 Linux 发行版中，可以使用 `docker search` 命令从 Docker Hub 搜索镜像。因为某些发行版（如 Red Hat Enterprise Linux）还会配置搜索自己的 registry，所以针对不同的 Docker 服务实现，`docker search` 命令的结果可能有所不同。

下面是使用 `docker search` 命令从 Docker Hub 查找镜像的例子：

```
# docker search ubuntu
INDEX       NAME                          DESCRIPTION     STARS OFFICIAL AUTOMATED
docker.io docker.io/ubuntu                Ubuntu is a...  1988  [OK]
docker.io docker.io/ubuntu-upstart        Upstart is an...28    [OK]
...
# docker search centos
INDEX       NAME                          DESCRIPTION     STARS OFFICIAL AUTOMATED
docker.io docker.io/centos                The official... 1149  [OK]
docker.io docker.io/blalor/centos         Bare-bones...   9             [OK]
...
# docker search fedora
INDEX       NAME                    DESCRIPTION          STARS OFFICIAL AUTOMATED
docker.io docker.io/fedora          Official Fed...      180   [OK]
docker.io docker.io/fedora/ssh                           19    [OK]
...
```

`docker search` 的输出按与镜像关联的星数进行了排序。拥有 Docker Hub 网站账户的用户可以为自己喜欢的镜像评星。除了镜像的描述外——这默认会被截断，输出还显示了该镜像是否由其名字（Fedora、Ubuntu、CentOS 等）所反映的项目官方创建，以及该镜像是否由自动化构建所创建。

如果你正创建自己的容器镜像，我建议你先从自己信任的 Linux 发行版的官方基础镜像开始。如果这个镜像没有包括你需要的所有软件包，你随时可以添加更多软件包（使用 `apt-get` 或 `yum` 命令）。Fedora、Ubuntu 和 CentOS 的基础容器镜像已经提前配置使用各自项目的软件仓库（获取 `deb` 或 `rpm` 软件包）。对于 Red Hat Enterprise Linux，rhel 的基础容器镜像会利用宿主机上启用的 Red Hat 软件仓库（假设你在 RHEL 系统上运行容器，而且该系统拥有有效的订阅）。

除了 Linux 基础镜像，Docker Hub 还有提前配置好的运行特定应用程序的容器镜像。有

时需要下载并修改某些软件来让应用程序按你想要的方式运行。

```
# docker search mysql
INDEX       NAME                          DESCRIPTION     STARS OFFICIAL AUTOMATED
docker.io   docker.io/mysql               MySQL is...     868   [OK]
docker.io   docker.io/orchardup/mysql                     40                [OK]
...
# docker search wordpress
INDEX       NAME                          DESCRIPTION     STARS OFFICIAL AUTOMATED
docker.io   docker.io/wordpress           The Word...     395   [OK]
docker.io   docker.io/ctlc/wordpress                      5                 [OK]

...
```

mysql 和 wordpress 的 repository 提供了来自 Docker Hub 的镜像，这些都是来自各自项目的正规软件。它们提供的容器已准备好运行 MySQL 数据库和 WordPress 富内容管理系统服务。

还能从 Docker Hub 上获取另一类容器，其介于基础操作系统容器和配置好的应用程序之间。这类容器旨在为开发或运行某类应用程序提供基础环境。下面是一些例子：

```
# docker search rails
INDEX       NAME                    DESCRIPTION  STARS OFFICIAL  AUTOMATED
docker.io   docker.io/rails         Rails is...  239   [OK]
docker.io   docker.io/lucio/rails   Latest Ruby  2                 [OK]
...
# docker search java
INDEX       NAME                    DESCRIPTION  STARS OFFICIAL  AUTOMATED
docker.io   docker.io/java          Java is...   257   [OK]
docker.io   docker.io/develar/java               9                 [OK]
...
# docker search golang
INDEX       NAME                    DESCRIPTION STARS OFFICIAL  AUTOMATED
docker.io   docker.io/golang        Go...       282   [OK]
docker.io   docker.io/google/golang             93                [OK]

...
```

可以使用 rails 镜像创建用于 Ruby on Rails 应用项目的 Dockerfile。对于运行 JavaScript 应用程序而言，node 镜像提供了名为 node 的官方容器镜像。java 镜像也是官方容器镜像，它既能够用来为 Java 应用程序提供运行环境，也能够提供 Java 应用程序的构建环境。golang 镜像提供的环境能够运行用 Google 的 Go 语言实现的应用程序。

如果要细化搜索，可以为 docker search 命令添加选项。例如：

```
# docker search -s 10 fedora
INDEX       NAME                    DESCRIPTION       STARS OFFICIAL AUTOMATED
docker.io   docker.io/fedora        Official Fed...   180   [OK]
docker.io   docker.io/fedora/ssh                      19    [OK]
# docker search --no-trunc=true mysql
INDEX       NAME                DESCRIPTION
```

```
                                    STARS   OFFICIAL   AUTOMATED
docker.io docker.io/mysql MySQL is a widely used, open-source
                          relational database management system (RDBMS)..
                                    868     [OK]
...
# docker search --automated=true centos
INDEX       NAME                        DESCRIPTION    STARS OFFICIAL AUTOMATED
docker.io   docker.io/blalor/centos     Bare-bones...  9                [OK]
```

在刚刚展示的例子中，-s 选项用来查找那些星数达到指定值的镜像（这里至少 10 颗星）。--no-trunc 选项告诉 docker search 不要截短描述字段。automate=true 选项让 docker search 只显示那些定期自动重新构建的镜像。

在为运行 docker 的系统查找镜像及其描述时，本节展示的这些 docker search 命令很有用处。然而，经常需要更多信息才能用好这些镜像。接下来的一节将介绍如何通过 Docker Hub 查找镜像的更多信息。

5.1.2　在 Docker Hub 上检索镜像

就多数容器镜像而言，仅仅是名称和简短的描述并不能让你了解需要知道的方方面面。有些镜像要用某种方式运行才能让镜像带着数据运行或是开启特定权限。对于存储在 Docker Hub 上的镜像，可以用 Web 浏览器检索 Docker Hub，通常会找到有关镜像的更多信息。

下面是用 Web 浏览器从 Docker Hub Registry（https://registry.hub.docker.com/）检索镜像的一些方法。

- **Registry 搜索框**：在 Regsitry 搜索框中输入查询项。你会看到镜像名或描述中包含该查询项的镜像列表。可以用多种方法对结果进行排序，这包括相关程度、创建日期、最新更新日期、评星数量等。和使用 docker search 命令进行搜索时一样，可以选择只列出官方镜像或者由自动构建所创建的镜像。选择想要进一步了解的镜像。图 5-1 展示的是用 Docker Hub Registry 搜索 Fedora 的例子。

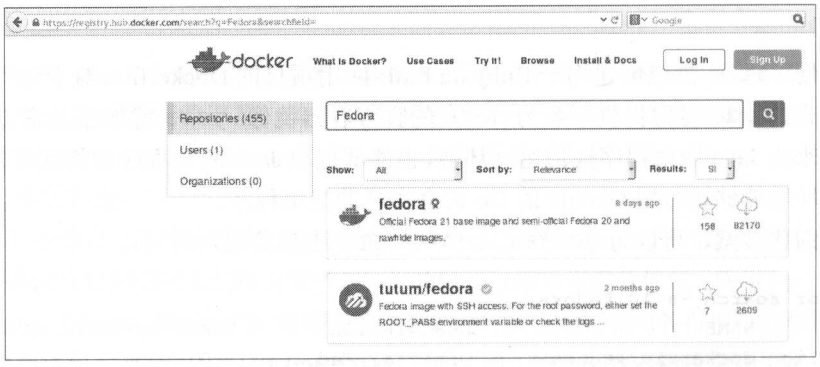


图 5-1　按关键词搜索 Docker Hub Registry 上的镜像

- **官方 repository**：从 Docker Hub Registry 页面上列出的官方 repository 中进行选择。

对获取特定 Linux 发行版（如 CentOS 或 Ubuntu）或开源项目（如 WordPress 或 MongoDB）的镜像而言，这是一个好办法。

一旦发现感兴趣的镜像，选中查看更为详细的信息。例如，在 Docker Hub Registry 的 WordPress 主页上（https://registry.hub.docker.com/_/wordpress/）有如何使用 WordPress 官方镜像的信息。如果缺少这些信息，就很难猜测用这个镜像都能干些什么。图 5-2 展示了 WordPress 官方镜像的页面。

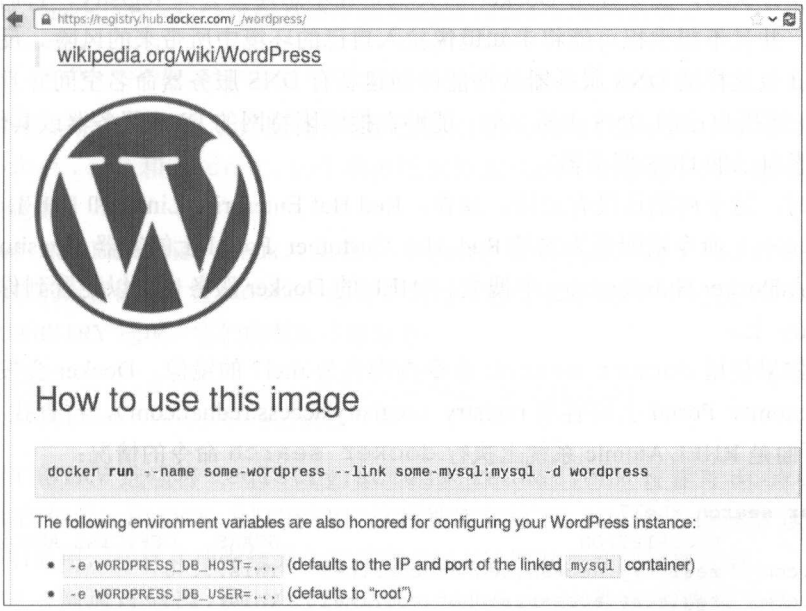


图 5-2 从 Docker Hub Registry 的镜像页面获取该镜像的详细信息

继续 wordpress 镜像的例子，在 Docker Hub Registry 的 WordPress 主页上能够找到下面这些镜像信息：

- 支持的镜像版本；
- 用来运行镜像的基本 `docker run` 命令；
- 配合 `docker run` 使用的环境变量，这些变量可以更改镜像设置；
- 这个容器镜像的 GitHub 页面的链接（https://github.com/docker-library/wordpress）；
- 用来构建该镜像的 Dockerfile。

如果要查找多个镜像的文档，可以直接访问 Docker Library（https://github.com/docker-library/doc），找到并点击镜像名来查看其文档。

5.2 其他 repository

某些 Docker 实现除了搜索 Docker Hub Registry 外也允许为 `docker search` 选择去哪

里寻找镜像。对一些 Docker 开发人员而言，这已经成为一个有争议的问题。

Docker 项目的初衷是：无论在哪里运行 Docker 格式的容器，Docker Hub Registry 命名空间都应该完全一样。因此，`docker search rhel` 和 `docker pull rhel` 会提供相同的结果，无论这些命令是在 Fedora、Ubuntu 上运行，还是在其他支持 Docker 的系统上运行。首先搜索的位置总是 Docker Hub Registry。争论的一方认为这应该类似因特网，如果在 Web 浏览器中输入 www.google.com，得到的应该总是同一个网站。

争论的反方认为，想要使用 Docker 的大型企业不想连接公共 registry。它们想完全控制自己的镜像，并且不想承担可能将未知镜像拉入自己的环境中所带来的风险。反方的一些人指出，像 bind 包这样的 DNS 服务器软件能够创建私有 DNS 服务器命名空间并且从不连接因特网，或者在解析自己的 DNS 主机名时，能够在搜索因特网的 DNS 服务器或其他 DNS 服务器之前先搜索自己的 DNS 服务器。

写本章时，这个问题还没有定论。现在，Red Hat Enterprise Linux 和 RHEL Atomic Host 系统中的 `docker` 命令被配置为先从 Red Hat Customer Portal 上的容器 repository 中搜索镜像，而后再从 Docker Hub Registry 中搜索。RHEL 的 Docker 服务也可以配置到你选择的任何 Docker registry 上。

例如，如果使用 `docker search` 命令查询名为 rhel7 的镜像，Docker 会按顺序先搜索 Red Hat Customer Portal 上的容器 registry（registry.access.redhat.com），再查找 Docker Hub Registry。下面是 RHEL Atomic 系统上执行 `docker search` 命令的情况：

```
# docker search rhel7
NAME          DESCRIPTION                              STARS   OFFICIAL AUTOMATED
redhat.com    registry.access.redhat.com/rhel          This... 0
redhat.com    registry.access.redhat.com/rhel6         This... 0
redhat.com    registry.access.redhat.com/rhel7         This... 0
```

在运行搜索命令前，在单独的终端窗口中跟踪 systemd 日志（使用 `journalctl` 命令），会看到RHEL Atomic 中的 `docker` 命令先查询了 Red Hat Customer Portal(`access.redhat.com`) 的 registry，之后又查询了 docker.io 的 registry。例如：

```
# journalctl -f -u docker
time="2015-04-14T23:09:22-04:00" level="info"
   msg="GET /v1.17/images/search?term=rhel7"
time="2015-04-14T23:09:22-04:00" level="info"
   msg="+job search(rhel7)"
time="2015-04-14T23:09:22-04:00" level="info"
   msg="+job resolve_repository(rhel7)"
time="2015-04-14T23:09:22-04:00" level="info"
   msg="-job resolve_repository(rhel7) = OK (0)"
endpoint.newEndpoint: starting with
   address=registry.access.redhat.com, secure=false
endpoint.newEndpoint: address after prefixing:
   registry.access.redhat.com
endpoint.newEndpoint:
```

```
    trimmedAddress=http://registry.access.redhat.com, version=
time="2015-04-14T23:09:23-04:00" level="info"
    msg="+job resolve_repository(docker.io/rhel7)"
time="2015-04-14T23:09:23-04:00" level="info"
    msg="-job resolve_repository(docker.io/rhel7) = OK (0)"
endpoint.newEndpoint: starting with
    address=https://index.docker.io/v1/, secure=true
endpoint.newEndpoint: address after prefixing:
    https://index.docker.io/v1/
endpoint.newEndpoint: trimmedAddress=https://index.docker.io,
    version=v1
endpoint.newEndpoint: terminating
```

我从每个日志消息中摘出了日期、时间戳以及主机名。你可以看到两个地址（registry.access.redhat.com 和 docker.io）都被用来搜索 rhel7。结果是，两个地址都给搜索返回了 rhel7 的镜像。

对于 Red Hat Enterprise Linux，从哪个 registry 搜索镜像取决于/etc/sysconfig/docker 文件中的配置信息。在 docker 服务启动时，它会在这个文件中查找 ADD_REGISTRY 和 BLOCK_REGISTRY 选项。它们的默认设置如下：

```
ADD_REGISTRY='--add-registry registry.access.redhat.com'
# BLOCK_REGISTRY='--block-registry'
```

ADD_REGISTRY 选项将 registry.access.redhat.com 设置为 docker search 命令首先查询的位置。BLOCK_REGISTRY 这行被注释掉了。因此，docker.io 这个默认启用的 registry 仍旧能用，并且其搜索结果会安排在 Red Hat 网站找到的结果之后。

下面这些示例通过修改/etc/sysconfig/docker 文件中的选项来改变 Red Hat Enterprise Linux 的本地 Docker 服务进行搜索时的返回结果。第一个示例中，如果不想搜索 Red Hat 的 registry，只要像下面这样把这一行注释掉，然后重启 Docker 服务就可以：

```
# ADD_REGISTRY='--add-registry registry.access.redhat.com'
```

如果要阻止从 Docker Registry Hub 搜索和下载镜像，可以取消注释并按如下方式修改 BLOCK_REGISTRY 选项：

```
BLOCK_REGISTRY='--block-registry docker.io'
```

要配置 Docker 服务搜索选定的 registry，可以添加一行 ADD_REGISTRY。例如，添加下面的配置能够搜索位于 myregistry.example.com（端口 5000）的 Docker registry。

```
ADD_REGISTRY='--add-registry myregistry.example.com:5000'
```

5.3 从 Registry 拉取镜像

想要从 Docker registry 下载镜像，从而能够在运行 Docker 服务的系统上访问到该镜

像，可以使用 docker pull 命令。当运行 docker run 命令所需要的镜像不在本地系统上时，docker pull 会被自动执行。尽管如此，一些人在使用镜像前更喜欢先把它拉取回来。原因之一是下载镜像有时要花一些时间，而你可能不想在运行的时候还要等待下载。

可以使用镜像的短名称（如 ubuntu 或 fedora）来指代想下载的镜像。但其他元素也能添加到镜像的短名称中。例如，假如有一个名称为 hangman 的镜像，当要拉取该镜像时，可能要明确其他信息以便获取想要的那个具体镜像，这些信息包括以下几项。

- **registry 名称**：这是镜像所在位置的名称和端口号。例如，myregistry.example.com:5000/hangman 标识了名为 hangman 的镜像，以及尝试从名为 myregistry.example.com 系统的 5000 端口拉取该镜像。
- **用户名**：添加 Docker Hub Registry 的用户名来标识将镜像推送到 registry 的用户。基本上，用户名类似 Docker Hub 的子目录。因此，从用户名为 cricket 的 respository 中拉取镜像时，拉取的名字会是 cricket/hangman。
- **标签**：标签是为同一个镜像添加多个名字的方法。当拉取一个名称中不包含标签的镜像时，隐含的标签是 latest。因此，如果拉取 cricket/hangman，下载的会是 cricket/hangman:latest。这个镜像在本地系统中可能是 cricket/hangman:latest、cricket/hangman:1.7 或者其他名字。如果镜像名还有其他版本，当拉取该镜像时必须明确指定该版本（如 cricket/hangman:1.5）。如果想使用 Fedora 或 Ubuntu 的特定版本的容器镜像，了解这些就至关重要了。（参见第 6 章，了解关于使用镜像标签的更多信息。）

下面是用来拉取镜像的命令行示例：

```
# docker pull cricket/hangman:1.7
Pulling repository cricket/hangman
be9a1bc2da8f: Download complete
1403322a81c5: Download complete
511136ea3c5a: Download complete
00a0c78eeb6d: Download complete
834629358fe2: Download complete

# docker pull cricket/hangman
Pulling repository cricket/hangman
1403322a81c5: Download complete
511136ea3c5a: Download complete
00a0c78eeb6d: Download complete
834629358fe2: Download complete

# docker images | grep hangman
cricket/hangman   1.7      be9a1bc2da8f   2 weeks ago      427.3 MB
cricket/hangman   latest   1403322a81c5   44 minutes ago   427.3 MB
cricket/hangman   1.9      1403322a81c5   44 minutes ago   427.3 MB
```

我们从输出可以看到两个镜像（be9a1bc2da8f 和 1403322a81c5）。一个镜像的标签是 1.7，

而另一个有两个标签——latest 和 1.9。如果只拉取最新镜像，就不会得到 1.7 的镜像（be9a1bc2da8f）。

要从 Docker Hub 之外的 registry 拉取镜像，可以将 IP 地址或主机名作为镜像名的一部分。在这个例子中，保存 hangman 镜像的系统地址是 192.168.0.118，docker-registry 服务运行在该系统的 TCP 5000 端口上：

```
# docker pull 192.168.0.118:5000/hangman
Pulling repository 192.168.0.118:5000/hangman
1403322a81c5: Download complete
511136ea3c5a: Download complete
00a0c78eeb6d: Download complete
834629358fe2: Download complete
```

在 docker pull 命令中，registry 名字的开头部分用来确定在哪里查找镜像，也可以将其替换为同一系统的主机名来拉取镜像：

```
# docker pull myregistry.example.com:5000/hangman
Pulling repository myregistry.example.com:5000/hangman
...
Status: Image is up to date for myregistry.example.com:5000/hangman
```

如前所述，使用短名字拉取镜像意味着想从 Docker Hub 上获取标签是 latest 的同名镜像。因此，运行 docker pull ubuntu 会拉取一组镜像，这是由于获取 latest 镜像时，该镜像可能具有多个标签：

```
# docker pull ubuntu
...
# docker images | grep ubuntu
ubuntu  14.04.2            d0955f21bf24   4 weeks ago   188.3 MB
ubuntu  latest             d0955f21bf24   4 weeks ago   188.3 MB
ubuntu  trusty             d0955f21bf24   4 weeks ago   188.3 MB
ubuntu  trusty-20150320    d0955f21bf24   4 weeks ago   188.3 MB
ubuntu  14.04              d0955f21bf24   4 weeks ago   188.3 MB
```

如果要测试 Ubuntu 容器的所有可用版本，可以添加-a 选项。记住，这可能会下载几个 GB 级的镜像。例如：

```
# docker pull -a ubuntu
Pulling repository ubuntu
...
# docker images | grep ubuntu
ubuntu 14.04.2             d0955f21bf24  4 weeks ago  188.3 MB
```

在这个例子中，使用-a 选项的结果是 17 个不同版本的 Ubuntu 容器镜像被拉取到本地系统。加上标签的话，大约会有 32 个不同的镜像名称。

5.4 保存和装载镜像

拉取并不是将镜像放入系统（以便 Docker 能够使用它们）的唯一方式。可以将本地系统的镜像保存到一个 tar 文件，然后复制并用某种方法将其传到另一个系统中，在那里装载它。

docker save 命令会保存与特定 respository 关联的所有镜像。只是将 repository 名提供给 docker save 命令的话，该 repository 的所有版本、标签和父层会以 tar 格式流输出到标准输出。另一个办法是，可以把输出保存到文件中。

下面的示例先从 Docker Hub 的 CentOS repository 下载所有镜像，之后将这些镜像保存成一个 tar 文件，最后用 scp 命令把这个 tar 文件复制到另一个系统中：

```
# docker save -o allcentos.tar centos
# du -sh allcentos.tar
1.6  G allcentos.tar
# scp allcentos.tar host2:/tmp
```

到这里，allcentos.tar 文件已被复制到 host2 系统的/tmp 目录下。现在，可以登录 host2，并按如下方法装载 allcentos.tar 文件：

```
# docker load -i /tmp/allcentos.tar
# docker images | grep centos
REPOSITORY TAG                 IMAGE ID      CREATED       VIRTUAL SIZE
centos     7                   0114405f9ff1  3 days ago    215.7 MB
centos     centos7             0114405f9ff1  3 days ago    215.7 MB
centos     latest              0114405f9ff1  3 days ago    215.7 MB
centos     7.1.1503            b58de3b24eb7  2 weeks ago   212.1 MB
centos     centos7.1.1503      b58de3b24eb7  2 weeks ago   212.1 MB
centos     5.11                2e4a66ce2189  6 weeks ago   284.2 MB
centos     centos5.11          2e4a66ce2189  6 weeks ago   284.2 MB
centos     6.6                 0bc55ae673f7  6 weeks ago   202.6 MB
...
```

从 docker images 的输出可以看到，来自 centos repository 的所有镜像现在都在新系统中。因此，如果想尝试用不同版本的 centos 来对应用程序进行容器化，现在系统中已经有多个版本的 centos 容器镜像。

如果只想保存和装载单个镜像，可以使用具体的镜像 ID 或镜像的全名，而不是使用镜像的短名字。镜像的全名可能包含 repository 的全名、用户名以及与该具体镜像关联的标签。下面是几个列子：

```
# docker save -o myhangman_1.7.tar cricket/hangman:1.7
# docker save -o myhangman_1.9.tar 1403322a81c5
```

至此，每个 tar 文件都包含了构成要保存的镜像所需的所有层。在将 tar 文件传输到另一个运行 Docker 服务的系统后，可按如下方式使用 docker load 命令装载它并查看结果：

```
# docker load -i myhangman_1.7.tar
# docker images | grep hangman
REPOSITORY          TAG     IMAGE ID        CREATED          VIRTUAL SIZE
cricket/hangman     1.7     be9a1bc2da8f    22 hours ago     427.3 MB
```

注意，这里只保存了特定的镜像，并将其装载到另一个运行 Docker 的系统中。该镜像已准备好使用。

5.5 小结

Docker Hub Registry 提供了对数以千计的 Docker 格式的镜像的访问。可以使用 `docker search` 命令查找 Docker Hub Registry 的镜像。当一个镜像 repository 中有你需要的镜像时，只要知道镜像 repository 的名字就能用 `docker pull` 命令从这个 repository 中将一个或多个镜像拉取到本地系统。

想将本地系统的镜像复制到另一个系统中时，可以使用 `docker save` 命令将单个镜像或整个 repository 的镜像保存为一个 tar 文件。然后，将这个 tar 文件复制到另一个系统中并使用 `docker load` 命令装载它，这样另一个系统的 Docker 就可以使用它了。下一章将深入如何处理镜像标签的细节。

第 6 章

为镜像添加标签

本章主要内容

- 命名容器镜像
- 为镜像分配标签
- 为镜像关联用户名
- 为镜像关联 respository 名

在使用 `docker build` 创建镜像之后，或者使用 `docker commit` 将容器提交给镜像之后，可以用 `docker tag` 为镜像添加额外的标签。这里使用了来自 Docker Hub 的名为 `ubuntu` 和 `fedora` 的镜像进行说明，下面是将标签添加到镜像名的末尾来进一步标识镜像的一些可能理由。

- **版本号**：版本号可以帮助识别在构建镜像时使用了软件的哪个版本。例如，`ubuntu` 镜像可以被标记为 15.04（表明 Ubuntu 的发布版本号），也可以标记为 15.04.2（标识出一个具体的修正版本）。同样，Fedora 项目为 `fedora` 的基础镜像附加了 20、21、22 这样的标签，来与 Fedora 的发行版相匹配。
- **版本名**：有时会为镜像的特定版本分配一个名字。例如，Ubuntu 15.04 也叫 vivid（对应名为 Vivid Vervet 的发布版）。Fedora 20 也为 `fedora:20` 镜像添加了名为 `heisenbug` 的标签。
- **latest**：当运行镜像却没有具体指定标签时，`docker` 命令会寻找标签为 `:latest` 的相应镜像。例如，执行 `docker run ubuntu` 命令会拉取并运行 `ubuntu:latest` 镜像。因此，为一个镜像附加 `latest` 标签就表明：如果请求时只用了基本的镜像名，那就会使用该镜像。

通过在镜像名开头部分附加的文本，可以标识出 registry——可以找到该镜像的地方（当运行 `docker pull` 命令时）或者可以保存该镜像的地方（运行 `docker push` 命令时）。下面是在镜像名的前面可以使用的内容。

- **registry 名和端口**：镜像标签最重要的用途之一就是添加 registry 名。在镜像名最开始的部分（以斜线分隔）添加 registry 名（以及可选的端口号），`docker pull` 或者

docker push 可以准确地知道去哪里获取你寻找的镜像或者将镜像保存到何处。当用户名被认定为 registry 名时，Docker 会去 docker.io hub 上寻找与该用户账户关联的 registry。

- **用户名**：在 docker.io registry 中创建用户账号后，可以将该账号的用户名放在镜像名的前面，标识该镜像是 docker.io 上对应用户的 registry 的一部分。例如，如果推送名为 docker.io/cricket/hangman 的镜像，hangman 镜像会被定向到 docker.io registry（Docker Hub）中用户 cricket 的私人存储区域。

本章将教你使用 docker tag 命令为现有镜像添加标签——其实就是用不同的名字标识镜像。

6.1 为镜像分配名称

在最初创建镜像的时候会为该 Docker 镜像指定一个镜像名。创建镜像有几种方法，下面是其中的几种。

- **构建镜像**：当从 Dockerfile 构建镜像时，可以为其分配一个名字。下面的示例使用当前目录中的 Dockerfile 构建镜像，并为其分配了名字 fedweb：

```
# docker build -t fedweb .
```

- **提交镜像**：在让镜像运行并按想要保留的方式做了一些更改后，可以使用 docker commit 命令将这些改变提交给一个容器镜像。在下面的示例中，我运行了 fedweb 镜像并将容器称为 newfedweb。之后我将 newfedweb 提交给名为 myfedweb 的新镜像：

```
# docker run -d -p 80:80 --name=newfedweb fedweb
# docker stop newfedweb
# docker commit -m "Web server with extra data" \
  -a "Chris Negus" newfedweb myfedweb
```

- **导出与导入镜像**：可以使用 docker export 命令将镜像以 tar 文件的形式保存到自己的文件系统中。将之前导出的 tar 文件导入系统时，可以为镜像指定一个名字。例如，我的朋友 Joe 将自己的容器（joefedweb）导出到 tar 文件中并将其发送给我。然后我导入了这个容器镜像。

```
# docker export joefedweb > joefedweb.tar
# cat joefedweb.tar | docker import - joefedweb
```

这时，在我的系统中有 3 个新镜像。注意，由于没有为镜像名添加标签，每个镜像都被赋予了“latest”作为其标签。例如：

```
# docker images  web
REPOSITORY  TAG      IMAGE ID       CREATED          VIRTUAL SIZE
```

```
joefedweb    latest    88abf604cfcc    12 minutes ago    526.6 MB
myfedweb     latest    8340292d5467    3 hours ago       706.5 MB
fedweb       latest    f583b458b447    4 hours ago       696.7 MB
```

接下来将用这 3 个镜像说明如何通过 docker tag 命令为镜像名添加标签和额外的 repository 信息，以使镜像更为有用。

6.2 为镜像分配标签

通过为镜像添加标签，可以更具体地描述镜像包含的内容。标签最常见的用法是为镜像添加版本信息。当有新版本的镜像时，它会成为镜像的“latest”实例，同时会保留镜像的其他标签（它们充当了同一个镜像的别名）——这些标签反映了镜像的其他定义信息。

比方说，你想为之前创建的镜像添加标签，来标识那些镜像的版本名和版本号。下面是具体做法的示例：

```
# docker tag fedweb fedweb:1.5
# docker tag fedweb fedweb:monkey
# docker images fedweb
REPOSITORY    TAG         IMAGE ID          CREATED           VIRTUAL SIZE
fedweb        1.5         f583b458b447      12 hours ago      696.7 MB
fedweb        latest      f583b458b447      12 hours ago      696.7 MB
fedweb        monkey      f583b458b447      12 hours ago      696.7 MB
```

因为这 3 个镜像的镜像 ID 完全相同(f583b458b447),所以可以判断出这 3 个 fedweb 镜像实际上是同一个镜像。接下来，假设你想将 myfedweb 作为 fedweb 的新版本，可以为 myfedweb 添加标签，标识该镜像还有 fedweb:1.7 和 fedweb:giraffe 这样的名字，并使其成为 fedweb 的最新（latest）版本。因为 fedweb:latest 已经存在，所以需要强制命名（-f）。

```
# docker tag myfedweb fedweb:1.7
# docker tag myfedweb fedweb:giraffe
# docker images *fedweb
REPOSITORY    TAG          IMAGE ID          CREATED          VIRTUAL SIZE
fedweb        1.5          f583b458b447      13 hours ago     696.7 MB
fedweb        latest       f583b458b447      13 hours ago     696.7 MB
fedweb        monkey       f583b458b447      13 hours ago     696.7 MB
fedweb        giraffe      8340292d5467      12 hours ago     706.5 MB
fedweb        1.7          8340292d5467      12 hours ago     706.5 MB
# docker tag myfedweb fedweb:latest
FATA[0000] Error response from daemon: Conflict: Tag latest is already set to
image f583b458b447..., if you want to replace it, please use -f option
# docker tag -f myfedweb fedweb:latest
# docker images *fedweb
REPOSITORY    TAG          IMAGE ID          CREATED          VIRTUAL SIZE
fedweb        latest       8340292d5467      12 hours ago     706.5 MB
fedweb        1.5          f583b458b447      13 hours ago     696.7 MB
```

```
fedweb          monkey          f583b458b447        13 hours ago        696.7 MB
fedweb          giraffe         8340292d5467        12 hours ago        706.5 MB
fedweb          1.7             8340292d5467        12 hours ago        706.5 MB
```

注意，为 myfedweb 强制命名（-f）fedweb:latest 镜像名和标签时，不仅使该镜像会成为用户请求 fedweb 时的默认镜像，而且解除了指定给原来镜像（f583b458b447）的 latest。另外，对于拥有多个名字的镜像，可以删除标签而不删除镜像本身。例如：

```
# docker rmi fedora:latest
Untagged: fedora:latest
```

6.3 为镜像分配 repository 名

镜像名称中的 *image_name:tag* 能够标识出这是什么样的镜像，附加在名称前面的信息还能够标识出存储镜像的 repository。当命名容器镜像时，repository 的信息是可选的，添加 repository 信息的语法如下：

```
[repository]:[port#]/[username]/image_name:tag
```

可以将 *username* 替换为 Docker Hub（docker.io）用户账号的名称。标识用户名可以很容易地将镜像推送到 Docker Hub 用户账号下的 respository 中。如果想将镜像推送到某个 registry 中——该 registry 运行在特定宿主机的（可选的）特定端口上，可以忽略用户名，并使用 IP 地址或宿主机名（连同可选的端口号）来标明私有 respository 的地址。

6.3.1 为镜像附加用户名

如果你有 Docker Hub 的用户账号，可以将 *username* 替换为用户账号的名称。例如，如果我在 docker.io 上有一个名为 cricket 的用户账号，我就可以将该用户名添加到那些我想推送到 docker.io 中的镜像上。按如下方法可以表明名为 fedweb:latest 的镜像与 docker.io 下的 cricket 相关：

```
# docker tag fedweb:latest cricket/fedweb:latest
# docker images *fedweb
REPOSITORY          TAG         IMAGE ID        CREATED          VIRTUAL SIZE
cricket/fedweb      latest      8340292d5467    12 hours ago     706.5 MB
fedweb              latest      8340292d5467    12 hours ago     706.5 MB
```

注意，现在这两个名字与同一个镜像 ID 关联。对于附加了用户名的镜像，可以将该镜像推送到 docker.io respository 中。继续假定用户名是 cricket，下面展示了将镜像推送到 docker.io respository 的对应账户时，如何进行鉴权：

```
# docker push cricket/fedweb
The push refers to a repository [cricket/fedweb] (len: 1)
Sending image list
```

```
Pushing repository cricket/fedweb (1 tags)
...
Pushing tag for rev [8340292d5467] on {https://cdn-registry-1.
docker.io/v1/repositories/cricket/fedweb/tags/1.5}
```

由于只为镜像添加了用户名，而不是一个单独的 repository 名，因此镜像被推送到 docker.io 上与 cricket 用户账号关联的 repository 中。如果以 cricket 登录 https://hub.docker.com/，可以看到已推送的镜像，如图 6-1 所示。

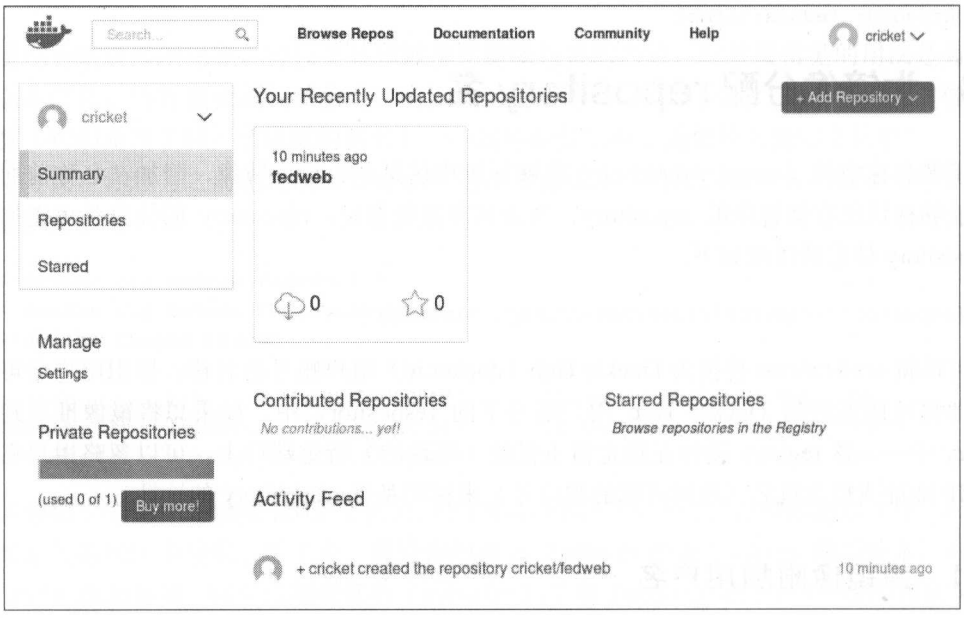


图 6-1 登录 docker.io 查看已推送的镜像

如果想用其他标签获取镜像，只要再给镜像打个标签并推送它就可以了。下面的示例为现有镜像添加了 `cricket/fedweb:monkey` 标签。注意，当推送新标签的镜像名到 docker.io 上的 cricket 的 registry 时，其显示那些构成镜像的层已经被推送到 docker.io 上了：

```
# docker push cricket/fedweb:monkey
The push refers to a repository [cricket/fedweb] (len: 1)
Sending image list
Pushing repository cricket/fedweb (1 tags)
511136ea3c5a: Image already pushed, skipping
00a0c78eeb6d: Image already pushed, skipping
...
```

回到 cricket 的账户，点击 Tags 标签页，会看到该账户中每个镜像以及该镜像的别名（标签）。在图 6-2 中，可以看到 `cricket/fedweb` 镜像有两个标签（1.5 和 monkey）。

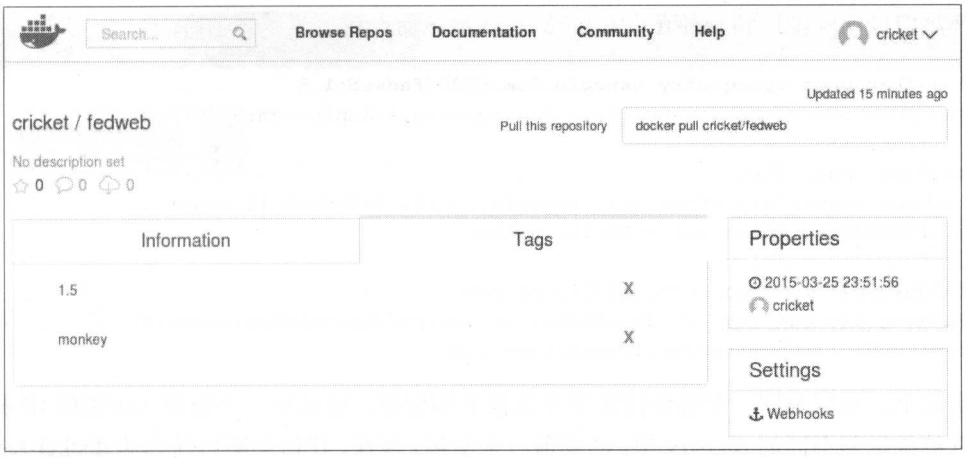


图 6-2 查看 docker.io 账户中为每个镜像分配的标签

由于我没有对这个镜像进行限制，任何想要使用 `cricket/fedweb` 镜像的人都可以用 `docker pull cricket/fedweb` 命令拉取该镜像，如图 6-2 中 Pull this respository 旁的文本框所示。所有人也都能够通过搜索 docker.io hub 上的 `cricket/fedweb` 找到该镜像。

```
# docker search cricket/fedweb
NAME                DESCRIPTION    STARS      OFFICIAL    AUTOMATED
cricket/fedweb                     0
```

随着容器条目展示在 docker.io 上，可以为镜像添加描述。在 Information 标签页中，选择编辑图标，可以为镜像添加简要说明和完整描述。在保存这些信息后，在搜索该镜像时会显示其简要说明。例如：

```
# docker search cricket/fedweb
NAME                DESCRIPTION                                STARS OFFICIAL...
cricket/fedweb      Test apache web server container 0
```

6.3.2 为镜像附加 repository 名

通过为镜像附加 repository 名，可以用 `docker push` 命令将该镜像直接推送到指定的 Docker registry 中。通过安装 docker-registry 软件包，启动 docker-registry 服务，并使用某个可访问系统的特定端口对外公开该服务，可以搭建自己的私有 Docker registry。

如果在名为 myregistry.example.com 的宿主机上搭建了 docker-registry 服务，并监听了默认端口（TCP 5000），就可以为镜像添加标签，从而将该镜像推送到这个 registry 中。例如，下面是为之前使用的 `fedweb:1.5` 镜像添加标签，以使其可以被推送到 myregistry.example.com 的方法。

```
# docker tag fedweb:1.5 myregistry.example.com:5000/fedweb:1.5
# docker images myregistry.example.com:5000/fedweb
REPOSITORY                        TAG  IMAGE ID       CREATED     VIRTUAL SIZE
myreg.example.com:5000/fedweb     1.5  f583b458b447   1 day ago   696.7 MB
```

要将镜像实际推送到 registry 中，可以输入如下命令：

```
# docker push myregistry.example.com:5000/fedweb:1.5
The push refers to a repository [myregistry.example.com:5000/fedweb]
   (len: 1)
Sending image list
Pushing repository myregistry.example.com:5000/fedweb (1 tags)
511136ea3c5a: Image successfully pushed
...
f583b458b447: Image successfully pushed
Pushing tag for rev [f583b458b447] on {http://myregistry.example.
com:5000/v1/repositories/fedweb/tags/1.5}
```

实际上，可以为同一个镜像指定多个镜像名和标签，这样同一个镜像（由镜像 ID 标识）就可以被推送到不同的 registry 中。回到前一个示例，现在，任何能够访问你的 docker-regsitry 系统的人都可以将该镜像拉取到自己的本地系统。例如：

```
# docker pull myregistry.example.com:5000/fedweb:1.5
Pulling repository myregistry.example.com:5000/fedweb
f583b458b447: Download complete
511136ea3c5a: Download complete
...
Status: Downloaded newer image for myregistry.example.com:5000/fedweb:1.5
```

6.4　小结

通过为 Docker 格式的容器镜像添加标签，不仅能够标识容器包含的内容，还能标识镜像保存在何处以及哪个用户账号的 repository 与其关联。通常会添加标识镜像中软件的版本号和版本名的标签。还可以为镜像添加宿主机名和 IP 地址这样的标签，这样就能向运行 docker-registry 服务的宿主机推送镜像以及从其拉取镜像。

第 7 章

探究容器

本章主要内容

- 检视镜像与容器的配置数据
- 查看镜像的历史
- 连接到容器进程以跟踪其所做的工作
- 在正在运行的容器中执行新进程
- 查看容器进程的日志信息
- 从正在运行的容器中复制文件

至此，对于容器，已经尝试了运行、推送、拉取、保存、装载和搜索。目前，这一切看起来有点儿像魔法。如果准备更深入地了解特定容器的处理过程，Docker 以及承载 Docker 运行的 Linux 系统有许多工具可以更细致地查看正在运行的容器。

本章将讲述如何在镜像运行前使用 `docker inspect` 命令查看镜像的配置。`docker inspect` 命令也可以用于查看容器的运行信息（如容器的进程 ID、网络接口和挂载的卷）和其他信息。

在仔细检视过镜像和容器后，可以使用其他命令进一步了解它们。有 `docker` 命令可以连接到容器的进程上、执行新进程、查看日志文件、复制文件，以及列出容器内的进程。

7.1 检视镜像和容器

每个镜像所附带的信息包含了默认配置、创建者以及有关镜像在何时、以何种方式创建的细节。在容器运行之后，容器会包含额外的信息，如容器的网络设置、容器是否尚在运行以及挂载卷的信息。

在镜像或容器上运行 `docker inspect` 命令可以查看与之关联的底层数据。在使用镜像和容器，或是用镜像和容器调试问题时，这能够提供帮助。例如，查看容器的 IP 地址可以知道客户端应用程序（如 Web 浏览器）去哪里寻找容器中运行的服务。

为了让输出更易于阅读，`docker inspect` 使用 JSON 格式显示。这使得查看每个单

独的键值对更容易，并且也更容易看出键值对如何契合到镜像或容器相关的信息结构中。

为了了解 docker inspect 是如何工作的，接下来的几节详细描述了如何查看与 fedora 镜像相关的底层数据，该镜像通过 python 命令和 SimpleHTTPServer 组件运行了一个简单的 Web 服务器。

7.2 检视镜像

第5章介绍了将Docker格式的容器镜像拉取到本地系统前，寻找和获取相关信息的方法。这些方法包括以下几种。

- **用 docker search 查找镜像**：通过该命令，能够用关键词检索 Docker Hub 和其他 registry，寻找满足需要的镜像。在查询结果中，能看到可用的镜像和 repository、镜像是官方的还是由自动构建生成的，以及镜像获得了多少评星。
- **浏览 Docker Hub 寻找镜像**：Docker Hub 网站（hub.docker.com）除了用于检索镜像外，还允许浏览可用的镜像，以及对它们排序。这个 registry 还常常显示镜像生成的方式以及如何使用这些镜像。
- **使用 docker 命令列出本地镜像**：当镜像位于本地系统中时，使用 docker images 命令可以查看镜像的基本信息。例如，可以查看镜像来自哪个 repository、镜像名和标签、镜像 ID、创建时间以及它的大小。

当选中镜像并将其拉取到本地系统后，在运行镜像前，你可能想检视一下镜像的底层信息。docker inspect 命令可以用来获取镜像及镜像创建方式的详细信息。

7.2.1 用 docker inspect 检视基础镜像

通过检视镜像可以大致了解镜像的创建时间、构建镜像所使用的 Docker 版本、对外开放的端口以及其他信息。下面是 docker inspect 命令的一个例子。该命令检视了最新的 ubuntu 容器。如果 ubuntu:latest 镜像尚未拉取到你的系统中，该命令将拉取对应的镜像并检视它。需要注意的是：每次拉取镜像，docker inspect 命令的输出以及镜像本身都有可能发生变化。我对输出进行了一些修整来突出所要展示的信息：

```
# docker inspect ubuntu:latest | less
[{
    "Architecture": "amd64",
    "Author": "",
    "Comment": "",
    "Config": {
        "AttachStderr": false,
        "AttachStdin": false,
        "AttachStdout": false,
        "Cmd": [
            "/bin/bash"
```

```
        ],
...
        "Env": [
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
        ],
...
        "NetworkDisabled": false,
...
    "Created": "2015-03-20T06:16:47.003636554Z",
    "DockerVersion": "1.4.1",
    "Id": "d0955f21bf24f5bfffd32d2d0bb669d0564701c271bc3dfc64cfc5adfdec2d07",
    "Os": "linux",
...
}
]
```

从这些输出中可以看到容器体系结构是 amd64 的（64 位 PC 兼容）。镜像创建时没有定义 Comment 和 Author。Config 部分设置了容器运行的环境。

如果运行时没有指定其他命令，默认会运行/bin/bash 命令。由于与标准输入、标准输出和标准错误的关联被设置为 false，因此，在运行该容器的默认 bash 命令时，需要在命令行中指定一些选项（如 docker run -it ubuntu）。Env 部分只设置了 PATH 变量，该变量定义了运行命令时用于查找命令的目录。NetworkDisable 设置为 false，表明容器内的网络接口应该被启动。

最后一部分显示了和容器创建相关的基本信息。通过这些信息能够了解到容器镜像创建的日期、创建容器镜像的 Docker 版本（1.4.1）、完整的容器 ID 以及操作系统（Linux）。

fedora 镜像的信息与此类似。下面是从检视最新 fedora 镜像的结果中选取的一些重要信息：

```
# docker inspect fedora:latest | less
[{
    "Architecture": "amd64",
    "Author": "Lokesh Mandvekar \u003clsm5@fedoraproject.org\u003e",
...
    "Created": "2014-12-31T22:33:27.395254797Z",
    "DockerVersion": "1.4.1",
    "Id": "834629358fe214f210b0ed606fba2c17827d7a46dd74bd3309afc2a103ad0e89",
    "Os": "linux",
}
]
```

与 ubuntu 镜像类似，fedora 镜像是基础镜像，因此没有什么具体配置。例如，与 ubuntu 基础镜像一样，fedora 镜像没有为网卡设置 MacAddress，只是简单设置网络接口要自动启动（"NetworkDisabled": false）。对于内存使用和需要的交换区也未做专门设置。这些基础镜像意在通用，以便用户基于这些镜像创建自己的容器和镜像时，可以按用户的要求进行调整。

7.2.2 用 docker inspect 检视应用程序镜像

如果说基础镜像意在通用，那么用于运行具体应用程序的镜像往往会对更多底层配置进行设置。wordpress 镜像就是包含了更多定制的例子（可以从 Docker Hub 获取）。下面是该镜像底层数据的示例：

```
# docker inspect wordpress:latest | less
...
"Config": {
...
    "Cmd": [
        "apache2-foreground"
    ],
...
    "Entrypoint": [
        "/entrypoint.sh"
    ],
    "Env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
        "PHP_INI_DIR=/usr/local/etc/php",
        "PHP_EXTRA_BUILD_DEPS=apache2-dev",
        "PHP_EXTRA_CONFIGURE_ARGS=--with-apxs2",
        "GPG_KEYS=6E4F6AB321FDC07F2C332E3AC2BF0BC433CFC8B3
➥0BD78B5F97500D450838F95DFE857D9A90D90EC1", "PHP_VERSION=5.6.8",
        "WORDPRESS_VERSION=4.2.1",
        "WORDPRESS_UPSTREAM_VERSION=4.2.1",
        "WORDPRESS_SHA1=c93a39be9911591b19a94743014be3585df0512f"
    ],
    "ExposedPorts": {
        "80/tcp": {}
    },
...
    "Volumes": {
        "/var/www/html": {}
    },
    "WorkingDir": "/var/www/html"
  },
"Created": "2015-04-30T05:58:47.912933967Z",
"DockerVersion": "1.6.0",
...
```

docker inspect 命令揭示出：与之前的基础镜像相比，更多容器特定的信息被加入到 wordpress 镜像中。在 Config 部分，位于容器根目录的 entrypoint.sh 脚本被设置为在容器运行时执行。apache2-foreground 脚本被定义为 entrypoint.sh 执行时的一个选项。

wordpress 容器定义了一组环境变量。这些变量设置了 shell 的 PATH 变量中的目录、包含 PHP ini 脚本的目录以及版本信息。ExposedPorts 变量将容器的 TCP 80 端口（HTTP 内容）开放给宿主机。Volumes 变量告诉 Docker 将宿主机上的/var/www/html 挂载到容器中，而后将 WorkingDir 设置为该目录，从而将该目录用作容器的工作目录。

与其他镜像一样，Created 这行给出了镜像的创建时间。DockerVersion 变量表明容器是为哪个版本的 Docker 设计的。

7.2.3 查看镜像的历史

可以在基础镜像上构建生成新镜像。每次在镜像上运行新命令，就会产生一个新的容器层。如果这些层与镜像保存在一起，之后可以通过 docker 命令查看这些信息。

有很好的理由将镜像分层保存，而不是将这些层压缩成一个镜像。例如，在同一台计算机上运行几个不同的容器，如果这些容器使用了相同的基础镜像（通过镜像 ID 来标识），可以节省存储空间。在基础镜像上可以添加某些开发库，而后可以添加一套标准服务。每一层重复的越多，节省的空间就越多。

要查看镜像的历史，只要在该镜像上执行 docker history 命令。示例如下：

```
# docker history ubuntu
IMAGE          CREATED       CREATED BY                                      SIZE
d0955f21bf24   6 weeks ago   /bin/sh -c #(nop) CMD [/bin/bash]               0 B
9fec74352904   6 weeks ago   /bin/sh -c sed -i 's/^#\s*...                   1.895 kB
a1a958a24818   6 weeks ago   /bin/sh -c echo '#!/bin/sh...                   194.5 kB
f3c84ac3a053   6 weeks ago   /bin/sh -c #(nop) ADD file:...                  192.5 MB
511136ea3c5a   23 months ago
```

可以看到，最初的镜像是 23 个月前创建的，6 周前该镜像被修改了 4 次。还可以看到生成镜像所运行的 shell 命令。之后的任何容器如果其使用了 IMAGE 列所列出的任何镜像层，就不必再拉取那个层，而只要使用系统中已经存在的这个层就可以了。

7.3 检视正在运行的容器

一旦容器运行起来，就可以通过查询该容器来获取查询原始镜像所能获得的大多数信息。除了镜像信息，还能看到 docker run 命令所设置的许多数据，以及其他在运行时添加到容器的配置设置。

docker inspect 的信息能够帮助检测正在运行的容器的问题，或者帮助理解容器的工作方式。可以对尚在运行的容器执行 docker inspect，也可以将 docker inspect 用在已经不再运行但尚未删除的容器上。换句话说，只要 docker ps 或 docker ps -a 命令能够列出的容器，都可以用 docker inspect 查看。

有时候，你只想把某个容器的底层配置导出来，还有些时候，你可能想选择特定的信息。docker inspect 命令允许指定名字并展示给定名字的值。

7.3.1 启动要检视的容器

为了实践容器检视，可以使用一些 docker run 选项来启动一个容器。对于这个示例，

我启动了 fedora 基础镜像并运行了 SimpleHTTPServer 模块，这会运行一个简单的 Web 服务器。

为了给 Web 服务器创建一些可分享的数据，我运行 echo 命令将一些文本放入到 index.html 文件中。之后，我启动了该 Web 服务器。下面是这两个命令：

```
# echo "Web Server Test: Successful" > /var/www/html/index.html
# docker run -d -p 8080:8080 --name="fed_web"        \
    --restart="on-failure:5" -w /var/www/html        \
    -v /var/www/html:/var/www/html                   \
    fedora python -m SimpleHTTPServer 8080
8f77610d10f2af42b158859ab7c689caf43829ebefda6ea7ac9bc5dce3890b6f
```

Web 服务器以守护进程（-d）的方式在后台运行，并且容器的 TCP 8080 端口被开放给宿主机。容器的名字被指定为 fed_web。通过设置--restart="on-failure:5"，当容器发生故障时，docker 会至多尝试 5 次来重启容器。（默认情况下，容器在故障时并不会尝试重启。）

Web 服务器的工作目录是/var/www/html。为了将数据放入容器的相应目录，我将宿主机的/var/www/html 目录挂载到容器中的相同目录。

使用的镜像是 fedora（其默认名为 fedora:latest）。之后，会用 python 命令运行 SimpleHTTPServer 模块，以便通过 TCP 8080 端口提供数据。要检验容器是否正在运行，可以在宿主机上键入下面的命令：

```
# curl -L localhost:8080
Web Server Test: Successful
# docker ps
CONTAINER ID  IMAGE           COMMAND                  CREATED
    STATUS        PORTS                          NAMES
8f77610d10f2  fedora:latest   "python -m SimpleHTT  5 minutes ago
    Up 5 minutes  0.0.0.0:8080->8080/tcp     fed_web
```

curl 命令查询本地的 TCP 8080 端口并返回我写入宿主机的 index.html 文件的内容（Web Server Test: Successful）。docker ps 命令显示：容器仍在运行，从宿主机 8080 端口请求的数据被定向到容器的 8080 端口。

容器已经启动并可以提供数据，是时候开始检视这个容器了。

7.3.2 检视整个容器配置

如果愿意，你可以将正在运行的容器的全部信息输出到屏幕上（STDOUT）。和镜像一样，可以使用 docker inspect 命令输出容器的底层配置信息。要查看这些内容的话，我建议用管道将输出连接到 more 命令上，或者将输出重定向到文件里来查看内容。

在下面的示例中，我对 docker inspect 的输出进行了拆解，重点展示了其中一些属性/值对：

```
# docker inspect fed_web
```

```
...
    "Args": [
        "-m",
        "SimpleHTTPServer",
        "8080"
    ],
    "Config": {
        "AttachStderr": false,
        "AttachStdin": false,
        "AttachStdout": false,
        "Cmd": [
            "python",
            "-m",
            "SimpleHTTPServer",
            "8080"
        ],
```

属性 Args 中保存了容器启动时传给 python 命令的 3 个参数。

Config 部分显示标准错误（AttachStDerr）、标准输入（AttachStdin）和标准输出（AttachStdout）并未关联到启动容器的终端会话上。如果要运行 bash shell，以便通过控制台直接与容器交互，这些设置应该被设为 true，而不是 false。

Config 部分的 Cmd 属性存放了命令（python）及传给该命令的所有参数（-m、SimpleHTTPServer 和 8080）。继续展示输出：

```
        "CpuShares": 0,
        "CpusetCpus": "",
        ...
        "Env": [
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
        ],
        "ExposedPorts": {
            "8080/tcp": {}
        },
        "Hostname": "8f77610d10f2",
```

如果系统中存在多个容器，你也许想要为每个容器分配 CPU 配额，以确保重要的容器获得更高比例的 CPU 时间（这里，CpuShares 并未获得 CPU 优先级）。如果设定了 CPUsetCpus（这里没有设置），容器将只在选中的 CPU 上运行。这两个值也可以通过 docker run 命令的-c 和--cpuset-cpus=""选项进行相应设置。

Env 部分包含 PATH 的设置，PATH 定义了在容器中执行命令时用来查找命令的目录。ExposedPorts 显示容器的 TCP 8080 端口开放到宿主机的相同端口上。由于命令行中没有设置主机名，容器 ID 的前 12 个字符被用作主机名。（docker run 的-h 选项可以为容器分配主机名。）输出继续：

```
        "Image": "fedora",
...
```

```
        "NetworkDisabled": false,
...
        "User": "",
...
        "WorkingDir": "/var/www/html"
    },
```

Image 属性显示使用的镜像名（本例中是 fedora）。NetworkDisabled 的默认设置 false 让容器中的网络从开始就处于启用状态。User 没有设置，如果设置了的话，容器内命令会使用指定的用户运行。例如，为 docker run 命令设置-u apache 选项，Web 服务器会以 apache 用户运行，而不是 root 用户。像之前提到的，工作目录（命令执行的目录）设置为 /var/www/html。在本示例中，Web 服务器从这里向外提供内容。输出继续：

```
"HostConfig": {
...
    "PortBindings": {
        "8080/tcp": [
            {
                "HostIp": "",
                "HostPort": "8080"
            }
        ]
    },
    "Privileged": false,
    "PublishAllPorts": false,
    "ReadonlyRootfs": false,
    "RestartPolicy": {
        "MaximumRetryCount": 5,
        "Name": "on-failure"
...
```

PortBindings 设置 TCP 8080 端口对宿主机上所有 IP 地址（HostIp）的 8080 端口开放。Privileged（false）表示容器没有访问其他容器或宿主机本身的权限，除了额外指定的几种方法（如明确地从宿主机挂载一个卷）。当 PublishAllPorts 设置为 false 时，容器中只有那些明确指定的端口（这里是 TCP 8080 端口）才能被宿主机上相同的端口访问。

ReadonlyRootfs 设置为 false，则 root 文件系统可以被写入。默认情况下，RestartPolicy 属性会设置为如果发生故障无需重启。在这个例子中，MaximumRetryCount 设置为 5。因此，Docker 在放弃尝试重启前，会尝试重启 5 次。输出继续：

```
"HostnamePath": "/var/lib/docker/containers/8f77610.../hostname",
"HostsPath": "/var/lib/docker/containers/8f77610.../hosts",
"Id": "8f77610d10...",
"Image": "834629358f...",
"LogPath": "/var/lib/docker/containers/8f776.../8f776...-json.log",
"MountLabel": "system_u:object_r:svirt_sandbox_file_t:s0:c157,c580",
"Name": "/fed_web",
```

一些要在容器内使用的标准 Linux 文件保存在宿主机上，并被挂载到容器上。这些文件以及其他信息保存在宿主机的/var/lib/docker/containers 目录中，具体位置是以容器 ID 命名的目录内。上面的示例中，我截短了容器 ID。

HostnamePath 设置了 hostname 文件的位置，容器的/etc/hostname 文件保存在这里。如果 docker run 命令没有明确设置主机名，那么主机名将被设置为容器 ID 的前 12 个字符。HostsPath 设置了/etc/hosts 文件的位置，该文件将容器的主机名与其 IP 关联在一起，同时还设置了 localhost 的 IP 地址以及 IPv6 地址。

LogPath 设定了与该容器关联的日志文件的位置。在/var/lib/docker/containers 目录下，每个容器对应一个子目录。子目录保存了对应容器的日志文件，文件名是容器 ID 加-json.log。要查看日志文件的日志消息，输入 docker logs fed_web 命令。

MountLabel 设置了 SELinux 上下文，这必须先设置在宿主机的文件上，而后该文件才能挂载到容器中。如果在/var/lib/docker/container/container_id/目录上运行 ls -lZ，你会发现 hostname、hosts 和 resolv.conf 文件的 SELinux 文件上下文完全相同，都是 MountLabel 属性设置的值（svirt_sandbox_file_t）。

Name 属性的内容是斜杠加容器名（/fed_web）。输出继续：

```
"NetworkSettings": {
    "Bridge": "docker0",
    "Gateway": "172.17.42.1",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "IPAddress": "172.17.0.9",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
    "LinkLocalIPv6Address": "fe80::42:acff:fe11:9",
    "LinkLocalIPv6PrefixLen": 64,
    "MacAddress": "02:42:ac:11:00:09",
    "NetNs": "",
    "PortMapping": null,
    "Ports": {
        "8080/tcp": [
            {
                "HostIp": "0.0.0.0",
                "HostPort": "8080"
            }
        ]
    }
},
```

Bridge 设定了网络名称（默认是 docker0），该网络为宿主机上的 Docker 容器提供了网络连接。该示例中，宿主机的网络地址是 172.17.42.1，正如 Gateway 属性所表明的。之后，IPv6 的几个属性都没有设置（因为默认情况下，Docker 不启用 IPv6 网络）。

MacAddress（02:42:ac:11:00:09）指出了容器内虚拟网卡的地址。Ports 设定显示

了 TCP 8080 端口到宿主机上所有 IP 地址（HostIp 是 0.0.0.0）及 TCP 8080 端口的分配。输出继续：

```
        "ProcessLabel": "system_u:system_r:svirt_lxc_net_t:s0:c157,c580",
        "ResolvConfPath": "/var/lib/docker/containers/8f77610d1.../resolv.conf",
        "State": {
            "Error": "",
            "ExitCode": 0,
            "FinishedAt": "0001-01-01T00:00:00Z",
            "OOMKilled": false,
            "Paused": false,
            "Pid": 10750,
            "Restarting": false,
            "Running": true,
            "StartedAt": "2015-05-04T22:23:09.12704039Z"
        },
        "Volumes": {
            "/var/www/html": "/var/www/html"
        },
        "VolumesMode": {
            "/var/www/html": {}
        },
        "VolumesRW": {
            "/var/www/html": true
        }
    }
    ]
```

ProcessLabel 展示了容器内进程运行时所用的 SELinux 安全上下文。如果在宿主机上执行 `ps -efZ | grep c157,c580`，可以看到一个正在运行的 python 命令——其带着 ProcessLabel 属性给出的安全上下文。还要注意，ProcessLabel 中“c157,c580”的部分与（前述）分配给挂载文件的安全上下文对应部分一致。如果系统中的 SELinux 处于启用状态，这一机制将防止普通容器访问其他容器的文件和资源。

State 属性提供了关于容器当前状态的信息。这里，由于容器正在运行，Running 属性的取值为 true。

输出信息的最后部分处理挂载卷。Volumes 属性展示了哪个本地目录被挂载到容器中。这个例子中是/var/www/html。之后，VolumesMode 属性设置了该目录上的权限，VolumesRW 设定了该目录是否只读。

7.3.3 检视单个容器属性

我们可以有选择性地展示容器的信息，无论该容器是正在运行、暂停还是已经停止。要做到这一点，只要在传递给 docker inspect 命令的--format 选项中指定你感兴趣的具体属性就可以了。

你不但可以获取容器的特定信息，还可以将这些信息传递给其他命令，针对这些信息采取行动。例如，你可以输入如下命令，检查容器的 IP 地址设置：

```
# docker inspect --format='{{.NetworkSettings.IPAddress}}'  fed_web
172.17.0.9
```

要检查 IP 地址是否处于运行状态以及是否能够访问，可以将输出传递给 ping 命令，如下：

```
# ping -c 2 $(docker inspect \
    --format='{{.NetworkSettings.IPAddress}}' fed_web)
PING 172.17.0.9 (172.17.0.9) 56(84) bytes of data.
64 bytes from 172.17.0.9: icmp_seq=1 ttl=64 time=0.127 ms
64 bytes from 172.17.0.9: icmp_seq=2 ttl=64 time=0.104 ms
--- 172.17.0.9 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.104/0.115/0.127/0.015 ms
```

如以上输出所示，该命令首先查询了 IP 地址，之后向容器发送了两个数据包并从容器接收到两个数据包。另一个例子，将 IP 地址传递给 curl 命令并尝试从 Web 服务获取内容。这里，我在命令行中添加了 8080 端口，这是 Web 服务监听的端口：

```
# curl -L $(docker inspect \
    --format='{{.NetworkSettings.IPAddress}}' fed_web):8080
Web Server Test: Successful
```

接下来介绍如何发现容器的不同类型的信息，以及在某些情况下，如何以这些信息作为其他命令的输入。

7.3.4 检视运行终端会话的容器

当启动容器运行 shell 终端会话时，可以通过另外的 shell 来检视该容器，只要该容器将标准输入（STDIN）、标准输出（STDOUT）和标准错误（STDERR）关联到这个 shell。之后，你会发现探究容器 shell 会话的新方式。

在 Docker 宿主机上打开两个 shell 会话，按照下面的步骤操作。在第一个 shell 中，输入下面的命令：

```
# docker run -it --name=bashtest fedora /bin/bash
bash-4.3# ls
bin  dev home lib64       media opt  root sbin sys usr
boot etc lib  lost+found  mnt   proc run  srv  tmp var
```

在启动名为 bashtest 的容器并打开一个 bash shell 后，在同一个宿主机上打开另一个 shell 并检视第一个 shell 与容器的标准输入、标准输出和标准错误关联的配置：

```
# docker inspect --format='{{.Config.AttachStdin}} \
```

```
    {{.Config.AttachStdout}} \
    {{.Config.AttachStderr}}' bashtest
true true true
```

如果想查看最初启动容器的 shell，甚至是与其进行交互，可以使用 docker attach 命令。在第二个 shell 中，输入如下命令：

```
# docker attach bashtest
bash-4.3# pwd
/
bash-4.3# ps
  PID TTY          TIME CMD
    1 ?        00:00:00 bash
    8 ?        00:00:00 ps
bash-4.3#
```

此时，可以在任何一个 shell 会话中输入。在一个 shell 会话中输入的东西将出现在另一个 shell 会话中。试验完成后，输入 exit 关闭 shell 并停止容器。在本示例中，我在一个 shell 会话中输入了 pwd 和 ps 命令，任何输入、输出和错误数据会在两个 shell 中出现。

图 7-1 展示了这两个 shell 会话的例子，第一个 shell 在前面。

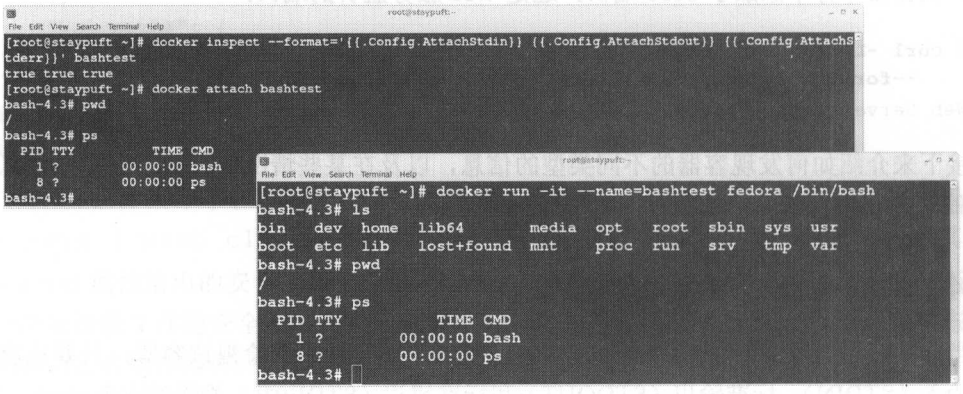


图 7-1 检视关联了 stdin、stdout 和 stderr 的容器

7.3.5 检视容器的内存和 CPU 限制

当你开始使用越来越多的容器，并且多个容器间还需要彼此交互，你会发现这些容器争夺资源。通过 docker run 选项，你可以限制一个容器可以使用的内存或交换区的数量。你还可以为容器设置 CPU 优先级并限制 CPU 的使用。

下面的示例分别列出了设置在容器上的内存和 CPU 限制。在默认情况下，运行容器时并不会对这些资源进行限制，因此我重新运行了 fed_web 容器，并添加了下面的选项：

- **--cpuset-cpus=0**：设置系统的第一个 CPU 执行来自容器的命令。
- **--cpu-shares=512**：设置容器能够得到的 CPU 时钟周期的比例。

- **--memory=1G**：限制容器能够使用的内存数量为 1 GB。
- **--memory-swap=2G**：限制容器能够使用的交换区的数量为 2 GB。

下面是逐个检视这些信息的方法。

```
# docker inspect --format='{{.Config.Cpuset}}' fed_web
0
# docker inspect --format='{{.Config.CpuShares}}' fed_web
512
# docker inspect --format='{{.Config.Memory}}' fed_web
1.073741824e+09
# docker inspect --format='{{.Config.MemorySwap}}' fed_web
2.147483648e+09
```

从上述输出中能够看到，容器被设置使用计算机的第一个 CPU（0）。容器的 CpuShares 设置为 512，默认是 1024。因此，这个容器将占用比通常更少的 CPU。

至于内存的使用，该容器被限制使用 1G（1.073741824e+09）的内存。对于交换区，容器可以使用 2G（2.147483648e+09）的交换区。

7.3.6 检视容器的 SELinux 上下文

在通常的 Linux 系统中，root 用户拥有一切并可以访问一切。当 Linux 系统实现了 SELinux 时（如 Fedora、RHEL 和其他高安全系统中那样），SELinux 可以限制运行进程所能访问的资源。这是通过让进程运行在特定的 SELinux 上下文中，并通过使用该上下文的 SELinux 来限制进程所能进行的访问来实现的。

Fedora、RHEL、CentOS 和类似系统在容器的底层数据中定义了两个 SELinux 标签（label），帮助限制容器访问宿主机上属于其他容器的内容。这两个标签是 MountLabel 和 ProcessLabel。下面是查看这些信息的一些方法：

```
# docker inspect fed_web | grep Label
"MountLabel": "system_u:object_r:svirt_sandbox_file_t:s0:c176,c430",
"ProcessLabel": "system_u:system_r:svirt_lxc_net_t:s0:c176,c430",
# docker inspect --format='{{.MountLabel}}' fed_web
system_u:object_r:svirt_sandbox_file_t:s0:c176,c430
# docker inspect --format='{{.ProcessLabel}}' fed_web
system_u:system_r:svirt_lxc_net_t:s0:c176,c430
```

MountLabel 的值是 SELinux 文件上下文，其设置在宿主机的文件上，这些文件可以挂载在该容器中。ProcessLabel 是用于该容器运行的进程的 SELinux 上下文。在 `fed_web` 容器的例子中，用于运行简单 Web 服务器的 `python` 命令使用该 ProcessLabel。

可以通过 `ps` 命令与 `-Z` 选项来查看进程的 SELinux 上下文。对于容器运行的进程可以挂载的文件，`ls` 命令与 `-Z` 选项可以查看这些文件的文件上下文。当容器仍处于运行状态时，可以取得容器运行的命令的进程 ID，而后可以将该 PID 传给 `ps` 命令，具体如下：

```
# docker inspect --format='{{.State.Pid}}' fed_web
32503
# ps -fZp 32503
LABEL                                          UID   PID   PPID  C  STIME
    TTY      TIME     CMD
system_u:system_r:svirt_lxc_net_t:s0:c176,c430  root  32503 1433  0  11:20
    ?        00:00:07 python -m SimpleHTTPServer 8080
# docker ps | grep fed_web
022ac55f6206 fedora:latest "python -m SimpleHTT 1 minute ago 1 minute
        0.0.0.0:8080->8080/tcp fed_web
# cd /var/lib/docker/containers/022ac55f6206*
# ls -Z hostname hosts resolv.conf | grep sandbox
-rw-r--r--. root root
    system_u:object_r:svirt_sandbox_file_t:s0:c176,c430 hostname
-rw-r--r--. root root
    system_u:object_r:svirt_sandbox_file_t:s0:c176,c430 hosts
-rw-r--r--. root root
    system_u:object_r:svirt_sandbox_file_t:s0:c176,c430 resolv.conf
```

之前的示例中，通过一开始的 docker inspect 命令， 查询到容器运行的进程的进程 ID（本示例中 PID 是 32503）。使用 ps 命令查看该进程 ID（-p 32503）的完整（-f）SELinux 上下文（-Z）列表，可以看到 SELinux 上下文与运行 Web 服务器（python 进程）的 ProcessLabel 中的上下文相匹配。

接下来查看该容器的 ID（022ac55f6206 是简短格式）并切换到位于/var/lib/docker/containers 的容器对应目录。使用 ls 列出 SELinux 上下文（-Z），可以看到 hostname、hosts 和 resolv.conf 文件上的上下文设置为容器中 MountLabel 的值。

关于 SELinux 上下文最后要说的一点是：从宿主机挂载到容器的每个文件，其文件上下文都含有 svirt_lxc_net_t。SELinux 上下文的最后一部分用来防止容器运行的进程访问属于其他容器的资源。在该示例中，上下文中的“c176,c430”是该容器特有的，其防止该容器访问宿主机的其他资源或其他容器的资源。

7.4 寻找探查容器的更多方法

现在，你的容器已经启动并运行，你已经完全检视过它的底层设置。使用 docker 命令的一些不同的子命令，你能够发现正在运行的容器的更多信息。

7.4.1 用 docker top 查看容器进程

一个容器通常只运行一个进程。然而，使用 docker exec 可以在容器中运行其他进程，最主要的，可以启动 shell 来查看容器中正在发生的事情。使用 docker top 可以查看容器中正在运行的所有进程。下面是具体的例子（读者看到的输出可能会有少许不同）：

```
# docker top fed_web
```

```
UID  PID   PPID C  STIME  TTY    TIME       CMD
root 12882 1433 0  23:17  pts/8  00:00:00   nsenter-exec --nspid
                                             32503 --console
                                            /dev/pts/8 -- /bin/bash
root 12883 12882 0 23:17  pts/8  00:00:00   /bin/bash
root 32503 1433  0 11:20  ?      00:00:07   python -m
                                             SimpleHTTPServer 8080
```

从输出中可以看到，容器中有 3 个进程正在运行。当该容器启动时，python 命令被运行。我用第二个终端打开/bin/bash，它通过启动宿主机上的 nsenter-exec 来打开容器所需的命名空间。docker top 显示宿主机上的这 3 个进程，连同其在宿主机上的进程 ID（PID）。

当在宿主机上执行 ps -ef 命令时，我们无法看到 nsenter-exec，只能看到 python 和/bin/bash 进程（以及刚刚执行的 ps -ef 命令）。注意，由于容器都有自己的进程表，因此容器内的进程 ID 有所不同：

```
# ps -ef
UID   PID   PPID   C STIME TTY   TIME       CMD
root    1      0   0 May05 ?     00:00:12   python -m SimpleHTTPServer 8080
root   15      0   0 06:38 ?     00:00:00   /bin/bash
root   20     15   0 06:38 ?     00:00:00   ps -ef
```

7.4.2 用 docker attach 与容器内服务进行交互

你可以关联到任何正在运行的容器来查看其运行的进程。如果容器仅是打开一个 bash（就像我之前做的那样），你可以通过其他进程关联一个 shell 来直接与那个 shell 进行交互。相同的交互也可以用于容器中运行的服务。

使用之前创建的 fed_web 示例，我可以连接到正在运行的容器，观察 Web 服务器对接收的请求的响应：

```
# docker attach fed_web
172.17.42.1 - - [06/May/2015 06:46:32] "GET / HTTP/1.1" 200 -
172.17.42.1 - - [06/May/2015 06:47:46] "GET / HTTP/1.1" 200 -
172.17.42.1 - - [06/May/2015 06:55:05] code 404, message File not found
172.17.42.1 - - [06/May/2015 06:55:05] "GET /badfile HTTP/1.1" 404 -
```

在刚刚展示的示例中，我运行了两个 curl localhost:8080 命令，前两行显示能够提供数据（默认 index.html 文件的内容）。第 3 行显示我尝试访问名为 badfile 的文件，而服务器上并没有该文件。当寻找/badfile 时，其结果是“file not found”和 404 状态码。

7.4.3 用 docker exec 在正在运行的容器中启动新进程

除了与容器运行的进程连接，实际上还可以启动一个新进程与容器进行交互。第 4 章有如何在正在运行的容器中启动 bash 的例子（例如，docker exec -it fed_web

/bin/bash)。然而，你可以在容器内运行你想要的任何命令（包括 yum 或者 apt-get 安装更多命令来运行）。

在这个例子中，我运行 docker exec 在 fed_web 容器中执行一条 yum 命令，该命令安装了 net-tools 软件包。接着，我执行了一些来自该软件包的命令，检查该容器网络接口的情况：

```
# docker exec -it fed_web yum install net-tools -y
Resolving Dependencies
--> Running transaction check
---> Package net-tools.x86_64 0:2.0-0.31.20141124git.fc21 will be installed
...
# docker exec -it fed_web route -n
Kernel IP routing table
Destination Gateway      Genmask      Flags Metric Ref Use Iface
0.0.0.0     172.17.42.1  0.0.0.0      UG    0      0     0 eth0
172.17.0.0  0.0.0.0      255.255.0.0 U     0      0     0 eth0
# docker exec -it fed_web netstat -tupln
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address Foreign Address State  PID/Program name
tcp        0      0 0.0.0.0:8080  0.0.0.0:*       LISTEN 1/python
```

net-tools 软件包安装后，我运行了来自该软件包的 route 和 netstat 命令。route 命令的输出显示：容器的默认网关是 172.17.42.1(宿主机 docker0 网络接口的 IP 地址)。netstat 命令显示：python 进程（该进程提供了 Web 服务）正在监听所有本地网络接口的 8080 端口（0.0.0.0.:8080)。

7.4.4 用 docker logs 查看容器进程的输出

除了连接到容器查看其实时处理的输出，还能回头查看容器处理的输出，无论容器正在运行还是已经停止（只要没删除)。在容器上运行 docker logs 会将容器处理的所有输出转储到屏幕上（STDOUT)，然后退出。

继续使用 fed_web 的例子，注意，这里的输出与用 docker attach 看到的输出完全相同，但它不会对输出持续关注，它会直接退出：

```
# docker logs fed_web
172.17.42.1 - - [06/May/2015 06:46:32] "GET / HTTP/1.1" 200 -
172.17.42.1 - - [06/May/2015 06:47:46] "GET / HTTP/1.1" 200 -
172.17.42.1 - - [06/May/2015 06:55:05] code 404, message File not found
172.17.42.1 - - [06/May/2015 06:55:05] "GET /badfile HTTP/1.1" 404 -
#
```

7.4.5 用 docker diff 查看容器的变化

Docker 会记录容器运行后容器内文件和目录发生的所有变化。使用 docker diff 命令

可以查看容器相对于最初运行的镜像的改变。在对 fed_web 容器做出一些修改后，下面是 docker diff 显示的这些变化：

```
# docker diff fed_web
A /var/www
A /var/www/html
C /root
C /tmp
D /root/anaconda-ks.cfg
A /tmp/anaconda-ks.cfg
...
```

输出显示新增了/var/www 和/var/www/html 目录（实际上，这些目录是从宿主机上挂载的）。我将 anaconda-ks.cfg 文件从/root 目录移动到/tmp 目录。这导致/root 目录和/tmp 目录发生了变化。它还显示/root/anaconda-ks.cfg 被删除以及/tmp/anaconda-ks.cfg 被增加。

在将容器提交为永久镜像前，最好使用 docker diff 检查一下容器的变化。

7.4.6 用 docker cp 从容器复制文件

也许有时候你想要看看容器中的文件，但并不想中断容器内正在进行的工作。一种办法是使用 docker cp 从容器中复制文件。要从正在运行的 fed_web 容器复制 index.html 文件，可以输入下面的命令：

```
# docker cp fed_web:/var/www/html/index.html /tmp
# cat /tmp/index.html
Web Server Test: Successful
```

7.5 小结

查看与镜像和容器相关的底层数据为了解它们的运作方式提供了一个良好的途径。在基础镜像上使用 docker inspect，可以看到运行该镜像的基础信息。检视用来运行具体应用程序的镜像能够揭示该容器的具体细节，如环境变量、入口点、挂载的卷以及工作目录。

使用 docker inspect 探究正在运行的容器的底层数据可以深入了解容器运行时发生的事情。对于每一个容器，可以看到正在运行的进程的 ID、网络接口、挂载的卷以及其他信息。

当容器正在运行时，还有许多其他方法发现关于容器的信息。使用 docker top 可以查看容器中正在运行的进程，使用 docker attach 可以连接到一个正在运行的容器进程，使用 docker exec 可以在正在运行的容器中执行新命令，使用 docker logs 可以查看容器正在运行的进程产生的日志消息，使用 docker cp 可以从容器中复制文件。

第 8 章

容器的启动、停止与重启

本章主要内容

- 停止和启动容器
- 重启容器
- 向容器发送信号
- 暂停容器与取消暂停
- 重命名容器
- 创建容器

我们已经运行了一些容器并对其内部做了一番探究。现在，我们已经准备好用容器做更多的事情。当容器镜像正在运行时，可以停止它、暂停它、杀死它或重启它。在容器运行之后（无论是容器停止了还是容器完成了运行），都可以再次启动那个容器实例。你也可以重命名重启并等着看它如何退出。

我会使用之前几章中运行的一些容器，展示如何使用不同的 `docker` 命令选项来操作创建的容器。之后，我会示范一种之后会用到的快速创建容器的方法。

8.1 停止和启动容器

当容器运行时，有几种不同的方式将容器暴露给宿主机系统。容器的端口可以暴露在宿主机上。容器内运行的命令可以将标准输入、标准输出和标准错误暴露给宿主机的 shell 会话。一旦容器停止（无论是其完成自己的任务，还是在其上运行了 `docker stop` 命令），它就会释放这些资源并以某种之后能够重新启动状态留存在系统中。

`docker start` 命令很简单。当容器是作为独立服务运行时，不需要任何命令选项。只有要交互式地运行来自容器的命令时，才需要为 `docker start` 命令提供选项。

除了通过分离的两步来停止和启动容器外，还可以使用 `docker restart` 命令来停止容器，随后立即使其再次运行。接下来的几节将介绍这些命令。

8.1.1 停止和启动分离的容器

我有一个名为testrun的镜像(参见第4章,了解我是如何创建它的),该镜像由fedora基础镜像和其上安装的 httpd 软件包组成。通过下面的命令，我使用分离选项（-d）来启动Web 服务器(httpd)，并使容器的/var/www 目录挂载到宿主机的/var/www 目录，将 TCP 80和 443 端口暴露给宿主机。(请确保宿主机上没有其他 Web 服务器正在运行并使用这些端口，否则容器运行将会失败。)

下面是运行 httpd 服务器的 docker run 命令，以及其他一些用来检验 httpd 服务器的命令：

```
# docker run -p 80:80 -p 443:443 -d --name=WebServer \
    -v /var/www/:/var/www/ testrun /usr/sbin/httpd -DFOREGROUND
# docker ps
CONTAINER ID   IMAGE           COMMAND                  CREATED
  STATUS          PORTS                                        NAMES
19a2b9483278   testrun:latest  "/usr/sbin/httpd -DF  2 hours ago
  Up 2 hours      0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp   WebServer
# netstat -tupln | grep -E "(:80)|(:443)"
tcp6  0  0 :::80        :::*        LISTEN        27194/docker-proxy
tcp6  0  0 :::443       :::*        LISTEN        27186/docker-proxy
# echo "The Web Server is Up" > /var/www/html/index.html
# curl http://localhost/
The Web Server is Up
```

docker ps 命令显示 WebServer 容器正在运行。运行 netstat 显示 docker-proxy 进程正在监听 TCP 80 和 443 端口。在将“The Web Server is Up”回显到/var/www/html/index.html 文件中后，运行 curl 命令显示 Web 服务器能够从挂载的宿主机目录下提供该文件。

现在，如果想要停止该容器，可以对容器 ID 或容器名使用 docker stop 命令。例如：

```
# docker stop WebServer
WebServer
# docker ps -a | head -n 2
CONTAINER ID   IMAGE          COMMAND              CREATED
  STATUS                  PORTS
    NAMES
19a2b9483278   testrun:latest "/usr/sbin/httpd -DF 2 hours ago
  Exited (0) 2 minutes ago 0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp
    WebServer
# netstat -tupln | grep -E "(:80)|(:443)"
# curl http://localhost/
curl: (7) Failed to connect to localhost port 80: Connection refused
```

运行 docker stop 后，该容器将不再出现在 docker ps 的输出中，但它仍会出现在 docker ps -a 的输出中。从输出可以看到，该容器在两分钟前（"2 minutes ago"）退

出了。/usr/sbin/httpd 进程不再运行，而容器的内容仍保存在本地系统中。

netstat 显示 docker-proxy 不再监听 80 和 443 端口。curl 命令显示 Web 服务器不再提供 index.html 文件的内容。

启动已停止的以分离模式运行（-d）的容器非常容易。保存的容器记得最初运行时的选项（暴露的端口、挂载的目录等）。容器运行时添加或修改的文件仍存在于容器中。因此，所要做的就是在该容器上运行 docker start。下面的例子再次启动已经退出的 WebServer 容器，然后检验其运行正常：

```
# docker start WebServer
WebServer
[root@fedora21 ~]# docker ps
CONTAINER ID  IMAGE          COMMAND                CREATED
  STATUS          PORTS                                          NAMES
19a2b9483278  testrun:latest "/usr/sbin/httpd -DF 23 hours ago
  Up 17 seconds   0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp WebServer
[root@fedora21 ~]# netstat -tupln | grep -E "(:80)|(:443)"
tcp6  0  0 :::80       :::*        LISTEN      3976/docker-proxy
tcp6  0  0 :::443      :::*        LISTEN      3968/docker-proxy
[root@fedora21 ~]# curl http://localhost/
The Web Server is Up
```

如你所见，WebServer 容器再次启动运行了。

8.1.2 启动和停止交互式容器

刚刚我们看到停止一个在后台运行服务的容器会发生什么。如果容器中运行的进程是交互式的，并且标准输入、标准输出和标准错误关联到本地的终端窗口，那么当在另一个窗口运行 docker stop 命令时，会话就关闭了，对打开会话的人而言相当唐突。

例如，像下面这样，打开一个 ubuntu 容器的 bash shell：

```
# docker run -it --name=bashbuntu ubuntu /bin/bash
root@10716cc23942:/
# pwd
/
```

接下来，在本地系统打开另一个 shell 并执行 docker stop 命令来停止容器。该容器从其启动的 shell 直接退出（exit 命令会在 docker stop 命令运行后自己弹出来，不需要输入）。

```
# docker stop bashbuntu
bashbuntu
# docker start -ai bashbuntu
```

在 docker stop 运行后，我再次启动了容器。为了使其再次以交互式的方式工作，我添加了下面两个选项。

- **-a**：连接选项（-a）将你的终端会话连接到容器运行的 bash shell 的标准输出和标准错误上。这让你可以看到容器的 bash shell 的输出。
- **-i**：交互选项（-i）将你的终端会话与容器内运行的 bash shell 的标准输入连接在一起。该选项使你能够将命令输入到容器内的 shell 中。

图 8-1 展示了从两个单独的 shell 会话启动容器、停止容器以及再次启动容器的一个示例。

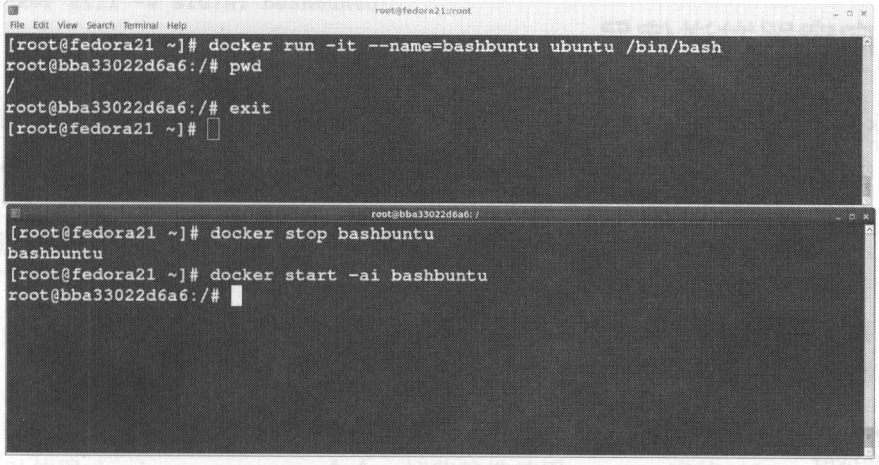


图 8-1 在停止交互式容器后，使用 docker start -ai 重新启动它

8.2 重启容器

可以不用通过分离的步骤来停止和启动容器，而是使用 docker restart 命令一次性完成这两件事。使用 docker restart 的优点是，能够尝试停止容器中运行的主进程，如果不能干净地停止，你可以告诉 docker restart：如果在若干秒没有停止的话就向容器发送 kill 信号。

尝试在 WebServer 容器上运行 docker restart 命令，将其停止并再次启动。有时，当容器进入奇怪的状态或者没有响应时，这样做就很好。重启一个运行服务的容器通常会让服务再次读取其配置文件。

下面是重启 WebServer 容器的例子：

```
# docker restart -t 30 WebServer
WebServer
# docker ps
CONTAINER ID  IMAGE          COMMAND                  CREATED
  STATUS        PORTS                                      NAMES
19a2b9483278  testrun:latest "/usr/sbin/httpd -DF 24 hours ago
  Up 3 seconds 0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp  WebServer
```

在这个例子中，容器立即停止紧接着就再次运行了。以防万一，我添加了-t 30 选项。

如果由于某种原因容器没有在 30 秒内停止，kill 信号就会发送到容器来杀死容器的主进程，从而杀死容器。

虽然 docker stop 和 docker restart 这两种方式可以发送信号来结束容器运行的第一个进程，但还有其他方法向容器显式地发送信号。docker kill 命令允许向选中的容器发送任何有效信号。

8.3　向容器发送信号

docker kill 命令向容器内运行的主进程发送 kill 信号（SIGKILL），这会立即杀死该进程，并杀死容器自身（以及容器内运行的其他进程）。如同 Linux 的 kill 命令，可以用 docker kill 命令向所选的容器发送任何信号。

通常，最好使用 docker stop 命令来停止容器，因为它会尝试干净地终止容器。然而，当容器由于某些原因挂起或者无法干净地终止，docker kill 一定能杀死容器。

如果我发现无法停止 WebServer 容器，我会用下面的命令杀死它。但要提醒一下，这可能会让容器无法再次启动。例如：

```
# docker kill WebServer
WebServer
# docker start WebServer
WebServer
# docker ps
CONTAINER ID   IMAGE   COMMAND   CREATED   STATUS   PORTS   NAMES
# docker log WebServer
httpd (pid 1) already running
```

杀死容器时没有适当地关闭它造成了 httpd 进程虽然停止却没有正确地被清理（在这个例子中，没有删除 PID 文件）。如果容器在杀死后无法再启动（或者系统没有正确地关闭），可以做的事情如下。

- **移除它**：多数情况下你只需删除掉无法使用的容器（docker rm）并执行新的 docker run 来从头开始运行容器。
- **保存并修复它**：在这种情况下，容器内尚有一些有价值的东西（可能是手动添加的数据或配置信息），可以将容器打包成镜像，将镜像导出为 tar 文件，解压 tar 文件，修复问题（这里是删掉 run/httpd/httpd.pid 文件），再次打包成 tar 文件，最后将 tar 文件导入系统中。
- **从脚本启动**：从脚本来启动服务，而不是直接从容器运行服务。在启动服务前，该脚本能够为运行服务进行需要的清理工作。在这个例子中，在启动 httpd 守护进程前，它将首先删除/run/httpd/httpd.pid 文件（如果有的话）。

因此，把话说清楚，若非必要，不要简单地杀死容器，因为当容器退出时，如果杀死容器没有进行适当的清理，它会使容器无法使用。即便如此，还可以用 docker kill 向容器

发送其他信号。

信号的联机帮助页列出了所有有效信号（输入 man 7 signal）。docker stop 向容器发送了一个terminate信号(SIGTERM)。在没有选项的情况下，docker kill 发送了kill信号(SIGKILL)。因此，向容器发送这两个信号时无需特别指定它们。下面的例子是向容器发送其他信号：

```
# docker kill -s SIGHUP WebServer
# docker kill -s SIGINT bashbuntu
```

当某些应用程序（如 httpd 服务）接收到 SIGHUP 信号，它们会重新读取其配置文件。对于不关掉服务的情况下更新 Web 服务器的配置，这是个好办法。SIGINT 信号向容器进程发送了一个键盘中断。当 SIGINT 信号发送给容器时，如果你正在另一个窗口运行该容器的 bash shell，其表现如同按下 Ctrl+C。你会在屏幕上看到^C，并且当前行会被中断。

进程接受的信号类型随应用程序的不同而不同。类似地，其响应每个信号的方式也各有不同。你应该检查要发送信号的应用程序以便了解其支持哪些信号。

8.4 暂停容器与取消暂停

使用 docker pause 命令可以暂停容器内运行的所有进程。要让容器的进程重新开始执行，可以使用 docker unpause 命令。这些特性使用 cgroups 冻结（freeze）特性来暂停和恢复所选容器的每个进程。

下面的例子是暂停和取消暂停前面示例中的 WebServer 容器：

```
# docker pause WebServer
WebServer
# docker ps
CONTAINER ID   IMAGE           COMMAND            CREATED
  STATUS                    PORTS
    NAMES
ad6851772d60  testrun:latest "/usr/sbin/httpd -DF About an hour ago
  Up About an hour (Paused) 0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp
    WebServer
# curl localhost
# docker unpause WebServer
WebServer
# curl localhost
The Web Server is Up
```

注意，从 docker ps 命令的输出可以看出容器已经被暂停。由于 WebServer 已经暂停，它不再响应对服务器内容的请求。在其取消暂停后，它再次正常响应。

8.5 等待容器的退出代码

使用 docker wait 可以确定一个运行的容器，之后等待它退出。当容器退出时，docker

wait 自己会退出，并打印最初容器的退出代码。要使用这个特性，首先启动 docker wait 命令，确定想查看退出代码的运行容器：

```
# docker wait WebServer
```

接下来，打开另一个 shell 并杀死 WebServer 容器，使其返回一个非零（不成功）的退出代码：

```
# docker kill WebServer
WebServer
```

回到 docker wait 命令，退出代码应该已经显示了。在这个例子中，退出代码是 137：

```
# docker wait WebServer
137
```

退出代码 137 表明 Web 服务器没有成功退出，而是被杀死的。

8.6 重命名容器

如果不喜欢容器的名字（无论其是运行中还是停止），可以改变。这不是什么难事：只要运行 docker rename 命令，并给出容器的旧名称以及想要的新名字。例如：

```
# docker ps
CONTAINER ID    IMAGE          COMMAND                  CREATED
 STATUS        PORTS                                       NAMES
38178d6e6ff8    testrun:latest "/usr/sbin/httpd -DF 7 minutes ago
 Up 7 minutes 0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp  WebServer
# docker rename WebServer HttpdServer
# docker ps
CONTAINER ID    IMAGE          COMMAND                  CREATED
 STATUS        PORTS                                       NAMES
38178d6e6ff8    testrun:latest "/usr/sbin/httpd -DF 7 minutes ago
 Up 9 minutes 0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp  HttpdServer
```

对比两个 docker ps 命令，可以看出容器的名字 WebServer 变成了 HttpdServer。

8.7 创建容器

创建容器（镜像的运行实例）的通常方法是使用 docker run 命令。通过 docker run 命令，只要不添加--rm 选项，容器退出后会被保存，以便能够再次运行它。如果只想创建容器而不立即运行它，可以使用 docker create 命令。

要使用 docker create，需要先确定要使用的镜像以及要在容器中运行的命令。当运行 docker create 时，它将直接保存结果容器，而不是在容器镜像中运行命令。

docker create 可以使用的选项与 docker run 的基本一致，但还是有几个例外。

例如，docker create 不能使用--rm，因为 docker create 的结果就是容器——这个容器被保存并为运行做好准备，如果把该容器删除就事与愿违了。

相同地，也不需要--detach=true（或-d）选项。之后启动容器时，其默认就是以分离模式运行的。如果想让容器以交互方式运行，需要添加-a 选项（将标准输出与标准错误定向到该进程）和-i 选项（将标准输入连接到该进程）。

由于 docker create 可以使用的选项与第 4 章描述的 docker run 的选项基本一致，可以参考第 4 章了解不同选项所起作用的更为完备的描述。这里，我提供了 docker create 的一些示例，以便读者可以看看该命令实际运行的情况。

例如，你想要一个运行 Boggle 游戏的容器，这个容器基于本书之前使用的 cricket/hangman 镜像，你想立即运行它。可以按下面的方法做：

```
# docker create --name=mybog -it cricket/hangman boggle
3dd0cdb06b377c020f93d160161d38e8b67cce49161707a6ec46c59d4af31b7f
# docker ps -a | head -n 2
CONTAINER ID  IMAGE                  COMMAND   CREATED
  STATUS                        PORTS     NAMES
3dd0cdb06b37  cricket/hangman:1.9   "boggle"  5 minutes ago
  Exited (0) 4 minutes ago                 mybog
# docker start -ai mybog
+---+---+---+---+    Type '?' for help
| Y | F | O | X |
+---+---+---+---+    2:51
| W | J | I | W |
+---+---+---+---+
| U | C | T | P |
+---+---+---+---+
| D | E | P | O |
+---+---+---+---+
fox
pit
duct
```

为了创建一个容器来运行 Boggle 游戏，我给容器添加了一个名字（mybog），并使用了-it 选项（明确容器要通过伪终端会话以交互方式来运行）。之后，我使用-a（连接标准输入）和-i（交互式）选项来启动这个容器。

8.8 小结

一旦容器保存到系统中(从 docker run 的执行返回或者运行一个新的 docker create 命令)，有许多方式与该容器打交道。可以使用 docker stop 来停止正在运行的容器，也可以用 docker pause 来暂停容器运行或者用 docker kill 来立即杀死容器运行。docker kill 命令还可以用来向容器发送其他信号，如 SIGINT（发送一个中断）或者 SIGUP（这会让某些应用程序重新读取配置文件）。

可以使用 docker restart 这一个命令来停止和启动容器，而不是用两个命令先停止，之后再启动它。操作容器的其他方式还包括 docker rename（重新命名容器）及 docker wait(等待容器退出并显示其退出代码)。如果要创建容器又不立即运行它，docker create 可以完成该工作。

第 9 章

配置容器的存储

本章主要内容

- 为容器添加存储
- 在容器间共享存储
- 管理宿主机的存储

Docker 容器镜像是要包含可以重复使用的应用程序。为了实现这一点，通常会挂载容器外的存储来保存容器数据。当使用 Docker 时，需要考虑以下两类存储。

- **管理容器的存储和卷**：为宿主机增加存储空间并不会自动让运行在宿主机上的容器获得更多的存储。然而，有多种方式可以在容器内部挂载和使用宿主机的存储卷。挂载到一个容器的卷也可以让其他容器使用。
- **管理宿主机上用于 Docker 的存储**：Docker 自身会使用宿主机的一个存储区域来管理该宿主机上的 Docker 镜像和容器，以及这些镜像和容器的元数据。当在一个或多个宿主机上搭建 Docker 时，管理员应该考虑什么样的最佳实践？

本章开始部分主要讲述容器存储的管理。接下来的部分会涵盖管理宿主机上的 Docker 存储的最佳途径。

9.1 管理容器的存储

Docker 镜像的创建者和使用者都想让镜像保持小巧。大镜像不利于下载，即使是在本地 repository 中也是如此。幸运的是，Docker 镜像仅需包含所选 Linux 发行版需要的软件包以及容器内运行的应用程序。

那么，容器如何在不提交镜像变更的情况下维护数据，并因此每当数据发生改变时无需改变镜像呢？解决方法是使用挂载外部卷。在应用程序中采用了这种方法，容器镜像不会发生改变。所有状态改变都会持久化到容器外部的存储中。

9.1.1 使用宿主机的卷

要在运行的时候将一个卷挂载到容器中，需要为 `docker run` 命令使用`-v` 选项。传递

给-v 选项的参数包括：要共享的宿主机目录（如/tmp/volume），紧接着是冒号，以及该目录在容器内的挂载点（如/data）。示例如下：

```
# docker run -v /tmp/volume:/data -d myappcontainer
```

但是，将卷挂载到容器中可能会将你暴露在一些潜在的危险之下。考虑下面的例子：

```
# docker run -v /etc:/data -d badappcontainer # 不要这么做
```

一个 Docker 容器认为自己有 root 权限。将/etc 或其他关键卷挂载到容器中是危险的。Docker 和容器技术应该为容器提供隔离。但这种情况下，隔离却被故意打破了。

在支持 Security Enhanced Linux（SELinux）的 Linux 系统中，当运行容器时，SELinux 提供了额外的保护和隔离。在 SELinux 增强打开的情况下，挂载文件或目录需要为文件或目录设置一个特定的 SELinux 标签。在下面的示例中，我在/tmp 目录下创建了一个新目录并将其挂载到容器中。在该容器中，我尝试在该目录中编辑文件。

```
# setenforce 1
# mkdir /tmp/vol
# docker run -v /tmp/vol:/data -it fedora:latest bash
bash-4.2#
```

当我通过容器内打开的 bash shell 尝试编辑或 touch 文件/data/myfile 时，SELinux 阻止我对该卷进行写操作。

```
bash-4.2# touch /data/myfile
touch: cannot touch '/data/myfile': Permission denied
bash-4.2# exit
```

要为容器明确地赋予修改挂载卷内容的权限，必须在该卷上使用:z 后缀来更改 SELinux 标签。例如：

```
# docker run -v /tmp/vol:/data:z -it fedora:latest bash
```

在容器内，我能够对该挂载目录进行修改：

```
bash-4.2# touch /data/myfile
bash-4.2# ls /data
myfile
```

奏效了！SELinux 通过强制我们明确对权限的修改来防止事故发生。在容器中，我们可以看到新文件也有适当的 SELinux 标签。

```
bash-4.2# ls -alZ /data/
drwxrwxr-x. 1000 1000 unconfined_u:object_r:svirt_sandbox_file_t:s0 .
drwxr-xr-x. root root system_u:object_r:svirt_sandbox_file_t:s0:c437,c645 ..
-rw-r--r--. root root system_u:object_r:svirt_sandbox_file_t:s0 myfile
```

当退出并查看宿主机中的/tmp/vol 目录时，我能够看到新文件及其数据：

```
bash-4.2# exit
$ ls /tmp/vol/
myfile
```

9.1.2 数据卷容器

使用 Docker 容器的优势之一是一个容器能够查看和使用其他容器中的卷。这种通常被称为数据卷容器的东西容许将源容器的卷共享给一个或多个目标容器。这其中的优势包括在多个容器间共享持久化存储并为挂载提供了一个抽象层。

当挂载一个卷到容器中时，给容器起个名字并为挂载点添加:z 后缀。这个容器就成为数据卷容器或源容器。下面是名为 datavol 容器的例子：

```
# docker run -v /tmp/vol:/data:z --name=datavol -it fedora:latest bash
```

在另一个 shell 中，运行新容器并使用--volumes-from=选项指定第一个容器名。例如：

```
# docker run --volumes-from=datavol -d fedora:latest touch /data/mydata
```

从仍为 datavol 容器打开的 bash shell 中可以看到已经创建的名为 mydata 的新文件。检查宿主机的/tmp/vol 目录，能看到那里也有新文件。

```
bash-4.2# ls /data
mydata
bash-4.2# exit
# ls /tmp/vol
mydata
```

此外，数据卷容器甚至不需要保持运行。如果退出 datavol 数据卷容器，仍能改变该卷。在目标容器中创建名为 mynewdata 的第三个文件。

```
# docker run --volumes-from=datavol -d fedora:latest touch /data/mynewdata
```

可以在宿主机中看到/tmp/vol/mynewdata 已经创建了。如果再次启动并连接到数据卷容器，可以看到/data 中也有相应改变。

```
# docker start datavol
datavol
# docker attach datavol
bash-4.2# ls /data
myfile mynewdata mydata
```

9.1.3 对挂载写保护

我已经展示了如何安全地挂载卷，也展示如何使用--volumes-from 与另一个容器共享卷。但是，如果需要一个容器中的卷，但又想让特定的卷不与其他容器（即其他试着用

--volumes-from 访问该卷的容器）共享会怎样？

Docker 使用:Z 后缀来达到此目的。注意是大写的 Z 而不是小写的 z。该后缀可以挂载一个卷在容器中使用，并防止其他容器使用它。

```
# mkdir /tmp/vol2
# docker run -v /tmp/vol:/data:z -v /tmp/vol2:/data2:Z --name=datavol \
    -it fedora:latest bash
bash-4.2# exit
# docker run --volumes-from=datavol -d fedora:latest touch /data2/mydata
touch: cannot touch '/data2/mydata': Permission denied
```

在刚刚展示的例子中，我能够对 datavol 容器内的/data 和/data2 目录（各自对应宿主机的/tmp/vol 和/tmp/vol2）进行写操作。但是，在第二个容器中，我能对名为 data1 的卷进行写操作，而无法对名为 data2 的卷进行写操作。这是使用:Z 后缀的缘故。

9.1.4 挂载设备

有时，能够挂载设备是非常有用的。例如，我已经运行了许多 Docker 容器，如果能够关联一个容器来检查日志文件以确认是否有错误发生，那会非常高效。

这样，我能够通过系统日志设备（/dev/log）使容器日志在宿主机上可见。示例如下：

```
# docker run -v /dev/log:/dev/log -i -t \
  fedora:latest logger "SYSLOG-TEST This is a test"
# journalctl -b | grep SYSLOG
May 05 18:08:41 myhost logger[3617]: SYSLOG-TEST This is a test
```

在刚刚的示例中，宿主机的/dev/log 设备被挂载到容器中。容器运行 logger 命令将消息发送给 systemd journal（通过/dev/log 设备）。容器在发送日志消息后退出。之后，在宿主机上执行 journalctl 命令会显示 logger 发送的消息。

9.1.5 挂载套接字

还有一些情况下挂载套接字也很有用。在容器内访问 TCP 通信就是这样的例子。

访问 Docker 守护进程套接字的具体例子是在容器内运行持续集成（continuous integration，CI）的 Docker 构建代理。挂载目录中出现 Dockerfile 文件会触发构建一个 Docker 镜像。Dockerfile 出现所触发的 CI 工具将在相同的宿主机上运行另一个 Docker 容器，来构建 Docker 镜像。

但该 Docker 容器并未运行在 CI Docker 容器中。因此，挂载/var/run/docker.sock 使容器内的 Docker 客户端能够在宿主机上运行 docker build 和 docker run 命令，尽管是从容器中发起调用的。这就避免了在 Docker 内运行 Docker。

下面的示例演示了 Docker 的几个特性和想法：

- 挂载套接字；

- 使用特权容器；
- 一个有趣的例子。

使用任意文本编辑器创建一个简单的 Dockerfile。下面是 Dockerfile 内容的一个示例：

```
FROM fedora:latest
MAINTAINER Doc Hand <dhand@thedocks.com>
RUN yum -y update; yum -y install systemd-libs docker; yum clean all
```

假设该 Dockerfile 在当前目录中，要用这个 Dockerfile 构建容器镜像，输入下面的命令：

```
# docker build -t myrepo/docker .
```

现在，运行 myrepo/docker 镜像，记住容器需要看到 Docker 守护进程正在监听的套接字：

```
# docker run -v /var/run/docker.sock:/var/run/docker.sock \
    -it myrepo/docker bash
```

现在在容器内运行一个 docker 命令。例如，如果尝试列出此时可用的镜像，这会失败：

```
# docker images
  2014/08/05 16:18:56 Get
   http:///var/run/docker.sock/v1.12/containers/json:
   dial unix /var/run/docker.sock: permission denied
```

前面的命令之所以失败是因为 SELinux 权限的缘故。想从容器内访问套接字，必须打开权限选项。这里，:z 或:Z 后缀无法提供帮助。要将特权模式设置为 true，按如下方式输入命令：

```
# docker run --privileged=true \
   -v /var/run/docker.sock:/var/run/docker.sock -it myrepo/docker bash
```

现在从容器内再运行一些 docker 命令。这些命令应该会成功：

```
bash-4.2# docker images
bash-4.2# docker ps
```

要了解更多特权容器和超级特权容器的信息，参见第 13 章。

9.2 Docker 宿主机的存储策略

Docker 镜像并非是一个镜像，它实际上是一组分层的镜像。新镜像可以从头开始制作，其通常被称为基础镜像。（在镜像名上运行 docker history 能够查看构成该镜像的层。）

通过在父镜像上添加层能够创建一个新镜像。层可以通过安装软件来添加，通过为镜像添加新目录或文件来添加，或者通过运行那些改变底层的命令来添加。

使用 Docker 时，有两种方法能够看到层。当我运行 docker pull 从 registry 拉取镜像

时，通常会看到为所请求的一个镜像拉取了多个镜像层。当运行 `docker build` 命令时，如果使用了 ADD、RUN 或 EXPOSE 这些 Dockerfile 命令，我看到不同的层被添加进来。这些层中的每一个都有唯一的名字，而且每一个都能在运行容器需要时被挂载。即便那些属于一次 `docker build` 的所谓的临时层，也被保存为最终命名镜像层的一部分。

如果我在 Fedora 基础镜像上安装了 Java 镜像，并在其上安装了 JBoss，图 9-1 显示了这些层。当运行基于该镜像的容器时，最上层代表了容器内可写的层，所有其他层都是不可变的。在这个例子中，容器运行在 Atomic Host 上。

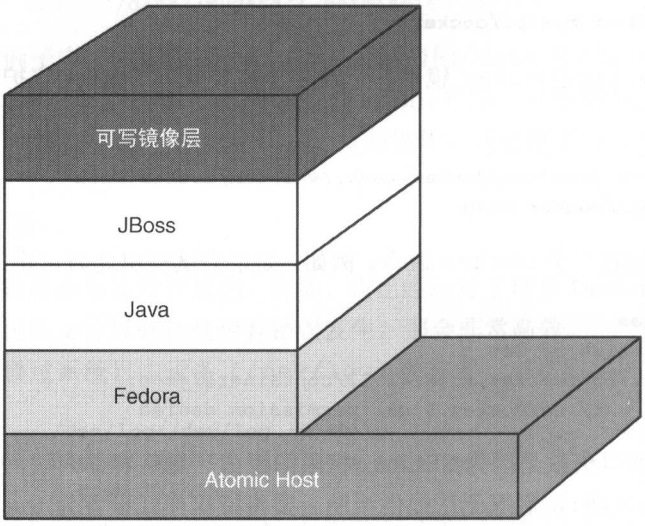


图 9-1 Docker 镜像是分层构建的，最上层是可写的

层使 Docker 和 Linux 容器区别于虚拟机（VM）。在一个虚拟机中，针对每种类型的虚拟机都有一个完整的镜像保存在机器上。如果像 Fedora 这样的公共组件要用于不同的虚拟机镜像中，每个镜像仍需将该公共组件包含在镜像中。对于 Docker 就仅需要特定版本 Fedora（如 Fedora 22）层的一个实例，而这个层会在 Java 和 JBoss 镜像之间共享，就像 Java 和 Fedora 镜像层会在 Java 和 JBoss 镜像之间共享一样。

Docker 使用存储后端的抽象来实现分层镜像。这使你能够管理完整镜像的各个层并将这些层在相关镜像间共享。因此，如果你的系统上已经有某个镜像的底下的层，那你就只需要最上面的层，`docker pull` 会只下载那个最上面的层。

Fedora、Red Hat Enterprise Linux、Fedora Atomic Host 以及 RHEL Atomic Host 使用 device-mapper 后端。这使 Docker 实现能使用 device-mapper 精简配置模块来实现层。

精简配置意味着：如果我有一个存储池，我就能夸大池的空间，因为我知道最初镜像使用可能比分配请求的要少。没有用的块不分配。

随着镜像需要更多存储，更多容量要从池里拿出来分配给那个特定的镜像。然而，池是有限的，当它可能在技术上过度分配时，管理员就要密切关注并在需要时为池添加更多存储。

还有其他存储后端实现，如 vfs、btrfs 和 aufs。在撰写本书时，它们要么被认为还不成熟（btrfs），要么上游内核还没有支持（aufs），要么就是不切实际（vfs，它在层之间不共享磁盘空间的使用）。这在未来可能会改变。

9.2.1 附加外部存储到 Docker 宿主机

在宿主机上，Docker 使用/var/lib/docker 卷来保存所有 Docker 镜像和容器。宿主机的 Docker repository 也包含在这个卷中。多数情况下，管理员不能精确地预测宿主机上的本地 Docker repository 将会变得多大。随着添加越来越多镜像并运行越来越多容器，这个卷会很快耗尽空间。

如果将 Docker 宿主机运行为一个虚拟机（如 RHEL Atomic Host），谨慎的做法是让 RHEL Atomic Host 镜像尽可能小。基础 RHEL Atomic Host 镜像确实很小。

如果你有一个 RHEL Atomic Host 的小镜像（如 8GB），这些空间不足以使用太多 Docker 镜像和容器。Docker 会很快把/var/lib/docker 目录（这里存放了 Docker 的所有镜像和元数据）填满。建议提供外部卷，并将该卷挂载到 Docker 宿主机的/var/lib/docker 目录。

在 Debian 和 Ubutun 上，可以修改/etc/default/docker 文件（添加 DOCKER_OPTS="-g /path/to/dir"）。但 Fedora 和 RHEL 不能使用这个方法，它们使用了 systemd。再者，对于 SELinux 标签，使用/var/lib/docker 目录是很重要的。

9.2.2 使用逻辑卷管理器扩展存储

以一个运行了 Atomic 虚拟机的 Fedora 宿主机开始，按照下列指令在宿主机上创建一个 LVM 卷，将其关联到 Atomic 虚拟机，并将其挂载到虚拟机的/var/lib/docker 目录。下面的指令假定 Docker 干净地运行在 Atomic 虚拟机上。当挂载新分区后，/var/lib/docker 目录下的所有重要 Docker 镜像或容器都能够复制到新分区并可以使用。

安全起见，如果有重要镜像、容器或其他元数据，应该备份它们，以防创建新磁盘空间时出什么问题。

（1）**创建 LVM 分区**。在 Fedora 宿主机上，对于这个例子，有一个/dev/sdb1 表示的 Linux LVM 类型（8e）的存储设备。我创建了一个 LVM 分区，该分区占用了整个 130 GB 的磁盘（我可以用 fdisk -l 命令查看）。下列命令的结果是名为/dev/docker_vg/mydocker 的 LVM 分区：

```
# fdisk -l /dev/sdb
Device     Boot Start       End     Sectors  Size Id Type
/dev/sdb1        2048 273672191   273670144  130G 8e Linux
# pvcreate /dev/sdb1
  Physical volume "/dev/sdb1" successfully created
# vgcreate docker_vg /dev/sdb1
```

```
Volume group "docker_vg" successfully created
# lvcreate -l 100%FREE -n mydocker docker_vg
Logical volume "mydocker" created.
```

（2）**创建文件系统**。在 Fedora 宿主机上，向新分区添加文件系统：

```
# mkfs.ext4 /dev/docker_vg/mydocker
```

（3）**将分区添加到 Atomic 虚拟机**。打开虚拟机管理器（virt-manager 命令），选择 Atomic 虚拟机，然后选择关闭它。选择 Show Virtual Hardware Details 按钮，选择 Add Hardware，选择 Storage，点击 Select Managed or Other Existing Storage，浏览并选择新的分区（/dev/docker_vg/mydocker）。确保 Bus type 是 IDE 以及 Device type 是 Disk Device，然后选择 Finish。

（4）**启动 Atomic 虚拟机**。启动 Atomic 虚拟机并以 root 用户登录。

（5）**停止 Docker 服务**。在 Atomic 系统中，按如下方式停止 Docker 服务：

```
# systemctl stop docker
```

（6）**复制 Docker 文件**。临时挂载新分区，从/var/lib/docker 处复制所有文件。这里，分区是/dev/sdb。例如：

```
# mkdir /mnt/tmp
# mount /dev/sdb /mnt/tmp
# cp -r /var/lib/docker/* /mnt/tmp/
# umount /mnt/tmp
```

（7）**获取 UUID**。在 Atomic 系统中，确定新存储设备的名字（这里，其显示为/dev/sdb）。然后，输入如下命令从设备获取 UUID：

```
# blkid /dev/sdb
/dev/sdb: UUID="9d73e64c-9422-459c-9677-27c2e8cbbc30" TYPE="ext4"
```

（8）**为挂载对分区进行设置**。将 UUID 添加到/etc/fstab 文件以便分区自动挂载到/var/lib/docker 目录。例如：

```
UUID=9d73e64c-9422-459c-9... /var/lib/docker ext4 defaults 1 1
```

（9）**挂载分区**。要将新分区挂载到/var/lib/docker 上，输入下面的命令：

```
# mount -a
```

（10）**重启 docker**。要重启 Docker 服务，输入下面的命令：

```
# systemctl start docker
```

添加存储的步骤已经完成。你可以再次使用 Docker。

9.3 小结

如果想用容器做更多事情——不只是运行几个简单的容器，能够添加外部存储就是至关重要的特性。将存储卷挂载到容器的特性使容器能够保持简单、可移植，同时这使得数据在容器外部可以被访问到。由于镜像和容器自身会消耗宿主机系统的空间，能够扩展宿主机的磁盘空间也是考虑因素之一。

要在运行镜像时为容器添加存储，可以使用`-v`选项确认要挂载宿主机上的哪个目录以及要将其挂载到容器中的哪个位置。还有一些`docker run`选项能够在多个容器间共享或限制卷的访问。

在宿主机系统上，Docker在`/var/lib/docker`目录中存储了镜像、容器和其他元数据。如果是在虚拟机上使用容器，可以关联宿主计算机的存储，将其挂载到虚拟机的`/var/lib/docker`目录上。用这种方式，可以增加存储在虚拟机中的Docker内容。

第 10 章

配置容器网络

本章主要内容

- 暴露网络接口的端口
- 将容器的端口映射到宿主机的端口
- 让容器绕过 Docker 网桥
- 为 Docker 服务修改网桥

Docker 为使用 Linux 容器（LXC）提供了一种便捷的方法。LXC 为每个容器提供了进程隔离和资源限制。而 Docker 在这类隔离之上提供了额外的层，使得容器有可移植性（镜像格式）并且易于使用（通过应用程序接口或命令行）。这种隔离也延伸至容器网络接口与宿主机系统的隔离方式上。

容器隔离的一部分会使每个容器认为它是自己机器的根进程。这就是有些人认为 Docker 容器是轻量级的虚拟机的原因。但 Docker 容器没有自己的独立内核，这一事实又使其有别于虚拟机。但 Docker 容器确实有着与虚拟机相似的行为。

Docker 产生这种感觉——容器是它自己的机器——的一种方式是，Docker 为容器进程提供了自己的 IP 地址。Docker 通过设置虚拟网络接口并将该接口桥接到宿主机的网络上来做这件事情。宿主机这边的网桥称为 `docker0`。该 `docker0` 网桥管理的每个容器都被分配了自己的 IP 地址。然而，Docker 仍提供了容器隔离。

如果容器要将基于 IP 的服务提供给其他容器或应用程序，它必须将该服务使用的端口暴露出来。例如，默认的 Apache Web 服务器容器应该暴露 80 和 443 端口，如下面的 Dockerfile 例子所示：

```
FROM fedora:22
MAINTAINER William Henry email ipbabble @ gmail dot com
# Update the system
RUN yum -y update; yum clean all
# Install httpd
RUN yum -y install httpd
EXPOSE 80 443
ENTRYPOINT /usr/sbin/httpd -DFOREGROUND
```

上面的 Dockerfile 使用 EXPOSE 关键字来定义从容器暴露的端口。使用 EXPOSE 关键字并不会将指定的端口立即暴露给宿主机系统上的其他容器或应用程序。这些实体要访问这个端口的话，还需要额外的步骤来连接或映射容器外部的端口。

- **暴露端口**：链接可以让容器访问同一机器上的其他容器暴露的端口。
- **映射端口**：映射提供了一种机制，可以将暴露的端口映射到宿主机的外部端口。

如果刚刚展示的 Dockerfile 位于当前目录，下面的命令会通过该文件构建一个名为 myrepo/fedora-httpd 的容器：

```
# docker build -t=myrepo/fedora-httpd .
```

在 10.1 节中，我将描述如何通过将容器链接在一起来让其他容器可以使用暴露的端口，然后展示如何将容器端口映射到宿主机端口，从而将它们暴露到宿主机外部。

10.1 将端口暴露给其他容器

使用前面构建的 myrepo/fedora-httpd 镜像（其在 80 和 443 端口暴露了它运行的 Web 服务器），我将另一个容器链接到它并让第二个容器使用暴露的 80 端口。首先，我运行了 myrepo/fedora-httpd 容器并给它起了 link-test 这个名字：

```
# docker run -d --name=link-test myrepo/fedora-httpd
```

注意，我是通过-d 参数以分离模式来运行这个容器的（而不是通过-it 参数在 TTY 上交互地运行）。

接着，我运行了第二个容器，我将其命名为 linked 并把它链接到 link-test。我将链接映射到名字 lt 上（link-test 的简称）。我在第二个容器内部使用这个名字访问第一个容器的 80 端口。

```
# docker run -it --link=link-test:lt --name=linked fedora:22 bash
```

我不是非得给第二个容器起名字，但这样做是一个好习惯。从第二个容器中运行 env（环境）命令，显示如下：

```
bash-4.2# env
HOSTNAME=d71eb38e62d5
TERM=xterm
LT_PORT_80_TCP=tcp://172.17.0.3:80
LT_PORT_80_TCP_PORT=80
LT_PORT_80_TCP_PROTO=tcp
LT_PORT=tcp://172.17.0.3:80
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
container_uuid=d71eb38e-62d5-0a15-902c-6901a4ed5d26
LT_NAME=/linked/lt
SHLVL=1
```

```
HOME=/root
LT_PORT_80_TCP_ADDR=172.17.0.3
_=/usr/bin/env
```

注意，有几个以 LT 开头的环境变量。这些变量名源自字符串 lt，这是我在 docker run 命令中分配给链接的名字。按照相同的方法，新容器可以访问第一个容器暴露的端口。

现在，我可以让 curl 命令利用两个容器之间的链接来显示名为 lt 或 link-test 的容器 80 端口上的 html 页面。这里，我只展示了开头几行：

```
bash-4.2# curl http://lt
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/
xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
   <title>Test Page for the Apache HTTP Server on Fedora</title>
...
```

链接是将容器联系到一起的强大工具。在 10.2 节中，我将建立一个场景。

10.2 将端口映射到宿主机外部

可以使用 docker run 命令的-p 选项来暴露容器的端口，以便通过宿主机的端口来访问它。通过该选项，可以将容器的端口暴露到宿主机的相同端口或不同端口。如果一个容器链接到另一个容器，宿主机的客户只能访问那些明确映射到该宿主机的端口。换句话说，即便容器链接使其他容器可以访问端口，但这并不会让相同的端口在宿主机系统中自动变得可用。

接下来的几节将解释如何将端口从容器映射到容器运行的宿主机。这几节还会展示端口映射对保护容器内的端口不被外部访问有何种影响。

10.2.1 从被链接的容器映射端口

图 10-1 使用两个已链接在一起的容器进行说明，这两个容器位于名为 H1 的宿主机上。宿主机 H1 有 docker0 这个默认的 Docker 网桥。一个端口在容器内暴露（容器 C1 的 8080 端口），一个端口被映射到宿主机的端口（容器 C2 的 80 端口）。下面的步骤描述了创建该配置的方式。

（1）**运行容器 C1**。这可能仅仅是一个应用程序，它通过 8080 端口暴露其基于 HTTP 的 UI。例如，它可能是一个 Jenkins 容器。在这个例子中，我用 python 模块运行了一个简单的 Web 服务器：

```
# docker run -d --name=C1 -w /var/www/html \
    -v /var/www/html:/var/www/html fedora:22 \
    /bin/python -m SimpleHTTPServer 8080
```

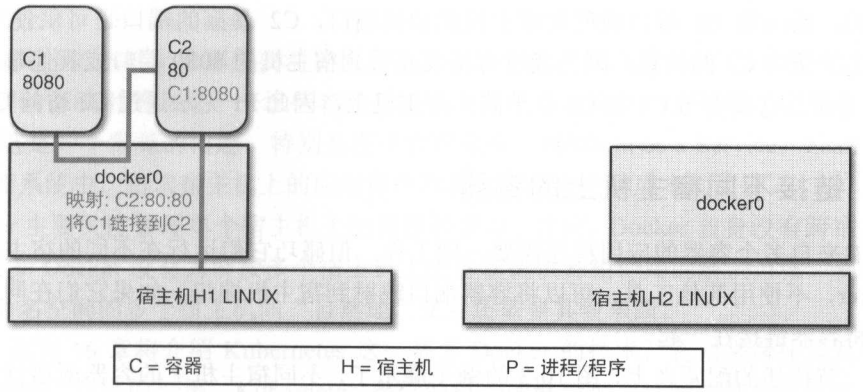


图 10-1 被连接的容器能够共享暴露的端口或将端口映射到宿主机

（2）**运行容器 C2**。接下来，我运行了容器 C2，并将该容器暴露的 80 端口映射到宿主机的 80 端口。我也将容器 C2 链接到容器 C1：

```
# docker run --name=C2 -d --link=C1:C1 -p 80:80 myrepo/fedora-httpd
```

（3）**检测从 C2 访问 C1 端口**。docker0 网桥让 C2 链接到 C1，也将 C2 的 80 端口映射到宿主机 H1 的 80 端口。如果通过 docker exec 命令运行 bash shell 进入 C2，我可以在 8080 端口上运行 curl，通过 C2 查看宿主机/var/www/html/中的内容：

```
# docker exec -i -t C2 bash
bash-4.2# curl http://C1:8080
The server is running.
```

（4）**检测从 H2 访问 H1 的端口**：现在，我在宿主机 H2 上启动了一个标准的 bash shell 进程 P1，该进程消费 C2 的 Web 页面，该页面暴露在 80 端口并被映射到 H1 的 80 端口。我只要用 H1 的 IP 地址运行 curl 命令即可。如同前一节中的例子，我应该会看到返回的 html 页面。图 10-2 说明了这个处理方式。

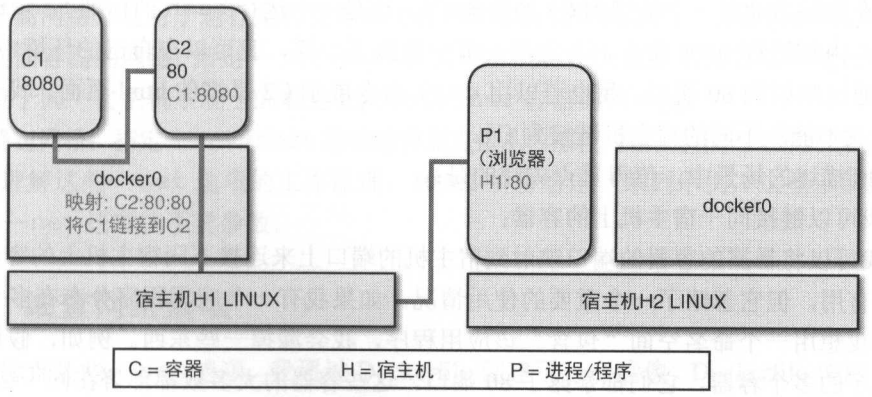


图 10-2 从 H2 的进程访问映射到 H1 上的容器端口

这说明，藉由将 80 端口映射到宿主机的相同端口，C2 暴露的端口是可供使用的。H2 的浏览器无法访问 C1 的信息，因为我没有将其映射到宿主机的 8080 端口或其他端口。由于 P1 不是容器而且它没有与 C1 和 C2 位于同一宿主机上，因此 P1 无法通过 C2 访问 C1。

10.2.2 链接不同宿主机上的容器

有可能来自多个容器的应用程序需要一同工作，但碰巧它们运行在不同的宿主机上。只使用 Docker，不使用其他工具，可以将容器端口映射到宿主机端口，如果它们在同一台机器上就可以将容器链接在一起。

在前一节描述的配置之上，图 10-3 的例子展示了，不同宿主机上的容器所运行的应用程序之间如何使用链接容器以及将端口映射到各自的宿主机上来建立联系。

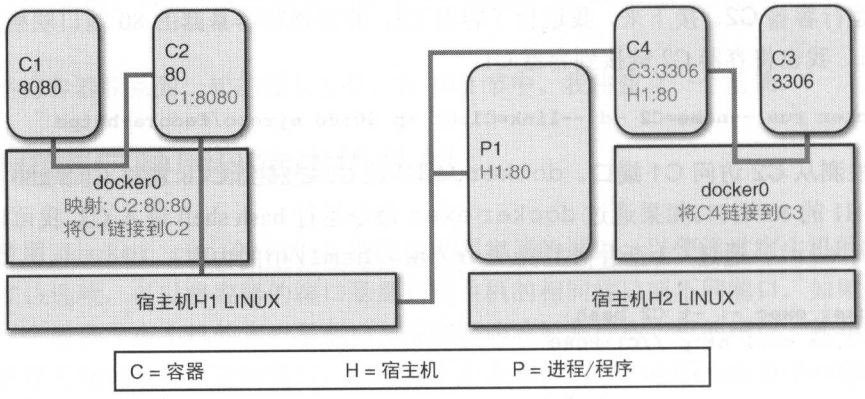


图 10-3 从不同宿主机上的容器访问端口

容器 C1 和容器 C2 仍旧运行在宿主机 H1 上，我在宿主机 H2 上启动了另一个容器 C3。这是一个暴露了 3306 端口的 MongoDB 容器。与 C1 相似，它既没有映射到宿主机端口，也没有链接到其他容器。

我现在能够启动另一个链接到 C3 的容器 C4。这是一个运行 `bash` 的标准 `fedora` 容器。如果在 C4 内部运行 `env` 命令，与之前一节中的例子一样，我可以在自己的环境中看到 C3 的链接。通过映射的 80 端口，我也可以用 `curl` 命令展示 C2 暴露的 html 页面。我无法访问 C1，因为我不能从不同的宿主机链接到 C1。

在刚刚描述的场景中，值得重点关注的是：

- 我可以链接同一宿主机上的容器；
- 我可以将暴露的容器的端口映射到宿主机的端口上来连接不同宿主机上的容器。

这很有用，但它忽略了一个重要的使用情况。如果我有一个应用程序分布在多个宿主机上，并且我想用一个命名空间“包含”该应用程序，我会漏掉一些东西。例如，假设我有多个应用程序的多个容器，它们都暴露了 80 端口。这些容器的大多数都部署在同一宿主机上。我不能将它们都映射到同一宿主机的 80 端口。我不得不将每个容器的 80 端口映射到宿主机

的某个端口上。

然而，其他宿主机上的那些要消费来自第一个宿主机上的容器80端口的信息的容器，不想负担了解动态映射的麻烦——这些容器要提供服务就必须进行动态映射。消费者只想读取80端口。这是一个常见的问题，特别是在平台即服务（Platform as a Service，PaaS）这样的服务供应商系统中，追求宿主机上的应用程序容器密度是效率和成本的关键。

Docker主要关注管理单个宿主机上的镜像和容器。目前，Docker自身没有跨宿主机的管理。换言之，Docker没有为链接多个宿主机上的容器提供开箱即用的解决方案。要让应用程序在相同命名空间的多个宿主机间“自然地”交互还需要其他东西。

第15章和16章将介绍Kubernetes这一基于Docker的技术，它用来编排涉及多个宿主机的多容器应用程序的部署。Kubernetes提供的机制能够管理容器间的宿主机和端口的动态映射。

10.3 docker0网桥的替代方法

尽管Docker的`docker0`网桥提供了对位于运行Docker的系统上的容器的网络访问，但不是非要使用这个网桥。这一节中我们会考虑两个案例：

- 容器使用`docker0`网桥的替代方法；
- 宿主机的Docker守护进程使用`docker0`的替代方法。

10.3.1 改变容器的网络模式

使用`--net`参数可以改变单个容器的网络模式。通过`--net`，可以使用默认网桥(`docker0`)、不同的网桥，或是根本不提供网络访问。下面的示例展示了使用`--net`选项的几种方式。

- **默认网桥**：指定`--net=bridge`会在名为`docker0`的Docker网桥上创建一个新的网络栈。这是默认行为。
- **无网络**：指定`--net=none`会通知Docker在不提供网络的情况下运行容器。容器会脱离网络。这对于运行某种交互实用程序（如计算器或游戏）会比较有用。
- **其他容器的网络**：指定`--net=mycontainer`会让Docker启动容器并使该容器重用位于同一宿主机系统上的名为`mycontainer`容器的网络栈。
- **绕过网桥**：指定`--net=host`意味着容器直接从内部使用宿主机的网络栈，绕过网桥。

为了理解这些`--net`选项的工作原理，10.3.2节将给出一系列示例，这些示例探索了能够提供给`--net`选项的不同参数。

10.3.2 细查网络选项

为了检查某些`--net`选项，我通过Dockerfile创建了一个镜像，Dockerfile使用`fedora`基础镜像并安装了net-tools软件包。下面是该Dockerfile的内容：

```
FROM fedora:22
MAINTAINER William Henry email ipbabble @ gmail.com
# Update the system
RUN yum -y update; yum clean all; yum -y install net-tools
ENTRYPOINT /bin/bash
```

保证 Dockerfile 位于当前目录，下面是构建该镜像的方法：

```
# docker build -t=nettools .
```

在开始运行容器前，我查看了宿主机包含 docker0 网桥的 IP 栈，看到了该网桥和宿主机的网络接口。我只展示了重要的接口并忽略了环回地址和运行在系统上的虚拟机的网络接口。

```
# ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
 inet 172.17.42.1 netmask 255.255.0.0 broadcast 0.0.0.0
 inet6 fe80::382b:6bff:fefb:9d8 prefixlen 64 scopeid 0x20<link>
 ether 5a:85:58:d1:ac:f3 txqueuelen 0 (Ethernet)
 RX packets 19427 bytes 1057038 (1.0 MiB)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 21384 bytes 266330243 (253.9 MiB)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
 inet 162.xxx.yyy.zzz netmask 255.255.255.0 broadcast 162.xxx.yyy.255
 inet6 fe80::aaaa:55ff:ccc:1 prefixlen 64 scopeid 0x20<link>
 ether 04:01:55:56:00:01 txqueuelen 1000 (Ethernet)
 RX packets 51651 bytes 512011975 (488.2 MiB)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 41427 bytes 3509038 (3.3 MiB)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

现在，我在没有指定--net 选项的情况下运行 nettools 镜像，容器因此从默认的 IP 地址池（172.17.0.0/16）中分配到 IP 地址。在这个例子中，它分配到的 IP 地址是 172.17.0.8：

```
# docker run -ti nettools
```

下面的 ifconfig 命令显示了标准 IP 栈，这个 IP 栈使用的网桥来自容器自己包含的网络接口。

```
# ifconfig
eth0: flags=67<UP,BROADCAST,RUNNING> mtu 1500
 inet 172.17.0.8 netmask 255.255.0.0 broadcast 0.0.0.0
 inet6 fe80::42:acff:fe11:8 prefixlen 64 scopeid 0x20<link>
 ether 02:42:ac:11:00:08 txqueuelen 0 (Ethernet)
 RX packets 6 bytes 508 (508.0 B)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 6 bytes 508 (508.0 B)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

```
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
 inet 127.0.0.1 netmask 255.0.0.0
 inet6 ::1 prefixlen 128 scopeid 0x10<host>
 loop txqueuelen 0 (Local Loopback)
 RX packets 0 bytes 0 (0.0 B)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 0 bytes 0 (0.0 B)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

注意，eth0 接口使用的是 docker0 网桥上的地址。这是默认行为：--net=bridge。现在，我使用--net=host 选项运行同一容器：

```
# docker run -it --net=host nettools
```

我在容器内部运行了 ifconfig，其展示了宿主机的网络接口，而不是容器内通常默认分配的独立网络接口：

```
# ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
 inet 172.17.42.1 netmask 255.255.0.0 broadcast 0.0.0.0
 inet6 fe80::382b:6bff:fefb:9d8 prefixlen 64 scopeid 0x20<link>
 ether 5a:85:58:d1:ac:f3 txqueuelen 0 (Ethernet)
 RX packets 19435 bytes 1057574 (1.0 MiB)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 21384 bytes 266330243 (253.9 MiB)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
 inet 162.243.151.77 netmask 255.255.255.0 broadcast 162.243.151.255
 inet6 fe80::601:55ff:fe56:1 prefixlen 64 scopeid 0x20<link>
 ether 04:01:55:56:00:01 txqueuelen 1000 (Ethernet)
 RX packets 51829 bytes 512026047 (488.3 MiB)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 41539 bytes 3524564 (3.3 MiB)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
...
```

注意，宿主机的所有接口都被展示出来(与之前一样，我只展示了 docker0 网桥和 eth0 接口)。因此，设置--net=host 会移除通常呈现的桥接层，而这对于提高性能会很有用。然而，当将这用于宿主机上的所有容器时，需要小心暴露的端口发生冲突。

这也很好地说明容器其实仅仅是宿主机内核之上的一个抽象层。它主要是在 Linux 内置的容器技术之上提供了一个层。

10.4 修改 Docker 网桥

如果默认的 docker0 网桥不适合组织的网络拓扑结构，还有一套机制来改变 Docker 守护进程使用的网桥。Docker 守护进程可以被启动并附加一个已存在的网桥。下面的命令将 Docker 守护进程附加到宿主机上名为 mybridge 的网桥上：

```
# /usr/bin/docker -d --selinux-enabled -b=mybridge
```

还可以关掉一个宿主机上所有容器的网络。例如：

```
# /usr/bin/docker -d --selinux-enabled -b=none
```

通常不必手动运行 Docker 守护进程。在使用 systemd 服务启动 Docker 的系统上（Fedora、RHEL、CentOS 和其他系统），在/etc/sysconfig/docker 文件中修改传递给 Docker 守护进程的选项。默认情况下，这些系统只是默认设置了--selinux-enabled。要添加之前所示的 mybridge 示例，编辑/etc/sysconfig/docker 中的 OPTIONS 行，使其看上去像下面这样：

```
OPTIONS='--selinux-enabled -b=mybridge'
```

下一次重启 Docker 服务时，它会获取新选项。例如：

```
# systemctl restart docker
# ps -ef | grep /usr/bin/docker

root 18540  1  0 13:06 ?    00:00:00 /usr/bin/docker -d
    --selinux-enabled -b=mybridge --insecure-registry localhost:5000
# docker run -ti nettools ifconfig eth0
[root@0f8e98ce321d /]# ifconfig eth0
eth0: flags=67<UP,BROADCAST,RUNNING>  mtu 1500
        inet 192.168.122.3  netmask 255.255.255.0  broadcast 0.0.0.0
```

在这个例子中，可以看到容器 nettools 从 192.168.122.0/24 中获取了一个 IP 地址，而不是从默认的 172.17.0.0/16 中。

10.5 小结

运行在 Docker 容器中的网络应用程序不同于直接运行在宿主机上的网络应用程序。因为，默认情况下容器所具有的网络接口不同于宿主机上的网络接口，要将网络端口暴露给其他容器或外部世界（通过宿主机端口）需要额外的工作。

构建 Docker 容器时，可以指定要暴露的端口。如果容器使用暴露的端口连接到其他容器，容器可以与暴露的端口传递信息。要让端口能在容器外部访问（或它链接的容器），可以将容器端口映射到宿主机的端口。

名为 docker0 的网桥是宿主机上的默认网桥，它为系统上的 Docker 容器提供网络连接。对于单独的容器，可以指定使用不同的网桥（或完全不用）。类似地，当启动 Docker 服务时，可以指定不同的网桥作为默认值。

Docker 容器的网络是可以独立于宿主机网络接口的一个特性，然而容器还包括其他特性来限制对宿主机权限进行访问。第 11 章会描述如何从系统中删除旧 Docker 容器和镜像，以及如何检测 Docker 是否正耗尽磁盘空间。

第 11 章

清理容器

本章主要内容

- 检查并分配磁盘空间
- 删除镜像
- 删除容器
- 保存前清理容器

每次拉取新镜像或者运行容器，都会消耗系统的空间。如果不关注的话，可能会吞噬掉所有的磁盘空间并导致 Docker（可能是整个系统）暂时无法使用。对此，Docker 提供了一些尽可能减少消耗的空间的方法，以及要清理容器时使用的一些清除不需要的容器的工具。

当然，节省计算机资源正是容器化的核心。因此，当 Docker 让创建和扩展容器变得容易的同时，它也让移除不再需要的镜像和容器变得容易。

本章将带读者了解一些 docker 命令，这些命令可以删除不再需要的镜像和容器。本章还会讲解 Docker 将其数据存储在何处，这样就能确保为其分配足够的空间来满足需要。这之后，读者可以阅读清理容器的方法，以便能够更高效地将容器保存为镜像，或是以某种方式运行容器——使其在退出时清理自己（docker run 的--rm 选项）。

11.1 为镜像和容器准备空间

Docker 将其数据保存在/var/lib/docker 目录结构中。如果 Docker 命令开始因为磁盘耗尽而失败，就需要确保该目录有足够的磁盘空间来处理需求。可以使用标准的 df 命令来做这件事：

```
# df -h /var/lib/docker
Filesystem              Size  Used Avail Use% Mounted on
/dev/mapper/rah-root    3.0G  2.3G  791M  26% /var
```

刚刚展示的例子是 RHEL Atomic 宿主机。如你所见，/var 目录总共只有 3.0 GB 空间可用——它们有可能被/var/lib/docker 使用。因为/dev/mapper/rah-root 是一个逻辑

卷，所以在需要时可以向这个逻辑卷添加更多空间。第 9 章描述了如何使用逻辑卷管理（LVM）向系统添加更多磁盘空间。

本章的重点放在了清理现存磁盘空间上。做到这一点的方法是列出系统上的镜像和容器，删除不再需要的那些。

11.2 删除镜像

镜像一旦被拉取或保存到系统上，就会一直存在直到被删除。对于不再使用的镜像，并不会自动清理。所以，如果磁盘空间有限，就需要时不时地删除一些不再想要的镜像。

要删除镜像，可以使用 `docker rmi` 命令以及要删除的本地镜像（或多个镜像）名。删除容器真的一点儿也不复杂，只是有些事情需要考虑。

- 尝试删除尚在使用的镜像（无论容器正在运行或暂停）会失败。可以无视这个行为，使用`-f` 选项强制删除。通常，在删除镜像前，最好停止该镜像运行的所有容器。
- 要删除的镜像必须明确。如果没有为想删除的镜像名指定标签，它假定要用`:latest` 标签。
- 如果一个容器镜像有多个名字（别名或额外的标签），`docker rmi` 只是删除标签而不是镜像本身。如果要求删除镜像时已没有其他名字，镜像会从系统上被物理删除。
- 当运行 `docker rmi` 命令时，删除物理镜像最为稳妥的方法是通过镜像 ID 来标识它。当这样做时，会看到在镜像自身被删除前所有与该镜像有关的标签都会被删除。
- 可以向 `docker rmi` 命令行添加多个镜像名或是用其他命令向 `docker rmi` 命令提供多个镜像名或镜像 ID，通过这种方式可以一次删除多个镜像。

接下来一节会描述如何从系统中删除单个镜像。

11.2.1 删除单个镜像

下面的例子说明了如何从本地系统删除镜像，包括如何处理前一节中描述的一些问题。对于这些示例，我从 Docker Hub 将所有 Ubuntu 镜像拉取到本地系统（`docker pull -a ubuntu`）。结果是超过 30 个独立镜像，其多数都有着多个名字。我使用 `docker images` 命令将它们展示出来：

```
# docker images | grep ubuntu
REPOSITORY          TAG                 IMAGE ID      CREATED      VIRTUAL SIZE
docker.io/ubuntu    14.10               d191563ad36b  7 days ago   194.4 MB
docker.io/ubuntu    utopic              d191563ad36b  7 days ago   194.4 MB
docker.io/ubuntu    utopic-20150427     d191563ad36b  7 days ago   194.4 MB
docker.io/ubuntu    latest              07f8e8c5e660  7 days ago   188.3 MB
docker.io/ubuntu    trusty              07f8e8c5e660  7 days ago   188.3 MB
docker.io/ubuntu    trusty-20150427     07f8e8c5e660  7 days ago   188.3 MB
docker.io/ubuntu    14.04               07f8e8c5e660  7 days ago   188.3 MB
docker.io/ubuntu    14.04.2             07f8e8c5e660  7 days ago   188.3 MB
```

```
docker.io/ubuntu trusty-20150320      b7cf8f0d9e82 2 weeks ago 188.3 MB
```

docker rmi 命令对镜像有两种不同的影响。如果删除一个镜像名，而这个镜像关联了多个名字，这个名字会被删除（取消标签），但镜像还在。如果用镜像名移除一个镜像而这是代表该镜像的最后一个名字，或者用镜像 ID 来删除镜像，镜像会从系统中被删除。下面是我尝试用不同方法删除这些 ubuntu 镜像时所发生的事情：

```
# docker rmi ubuntu
Untagged: ubuntu:latest
# docker rmi ubuntu
Error response from daemon: No such image: ubuntu:latest
# docker rmi ubuntu:utopic
Untagged: ubuntu:utopic
# docker rmi docker.io/ubuntu:utopic-20150427
Untagged: docker.io/ubuntu:utopic-20150427
# docker rmi 07f8e8c5e660
Untagged: docker.io/ubuntu:14.04
Untagged: docker.io/ubuntu:14.04.2
Untagged: docker.io/ubuntu:trusty
Untagged: docker.io/ubuntu:trusty-20150427
Deleted: 07f8e8c5e66084bef8f84887785...
Deleted: 37bea4ee0c816e3a3fa025f3612...
Deleted: a82efea989f94b1d9fac76e26e3...
Deleted: e9e06b06e14c2f7d8df0251e3bb...
```

名为ubuntu的镜像关联了多个别名。在第一个例子中，由于没有附加标签，docker rmi 假定你指的是 ubuntu:latest，并只删除该标签。注意，再次运行相同的命令并不会牵涉其他 ubuntu 镜像（它依然假定你指的是 ubuntu:latest）。接下来的例子中，你可以向基础镜像名添加任何标签来删除这个具体的镜像名（ubuntu:utopic）。虽然不需要，但你也可以添加 registry 名，解除镜像的这个标签名（docker.io/ubuntu:utopic-20150427）。

到目前为止的所有这些例子，由于输入的名字并不是本地系统上该镜像最后一个存在的名字，因此镜像只是被解除标签而没有被物理删除。物理删除镜像最简单的方法是用镜像 ID 来删除它（就像用 docker rmi 07f8e8c5e660 所做的）。而下面这个例子中，删除与镜像关联的最后一个名字也会将镜像自身删除。

```
# docker rmi docker.io/ubuntu:14.10
Untagged: docker.io/ubuntu:14.10
Deleted: d191563ad36befdc3322d27...
Deleted: 8d07608668f6d6265fc0d4b...
Deleted: 14975cc0f2bcfc529f862ef...
Deleted: 9802b3b654ece46f8e09379...
```

谨记，镜像常常由多个层组成（每个层都被 Docker 当作一个镜像）。因此，尽管你可能只是要删除一个镜像，却有多个层被删除了。

11.2.2 删除多个镜像

如果只是想清除本地系统的所有镜像，可以列出所有现存的镜像 ID 并将这个列表提供给 docker rmi 命令。获取镜像 ID 列表最简单的方法是使用 docker images -q 命令：

```
# docker images -q
b7cf8f0d9e82
1403322a81c5
1403322a81c5
9a8ad4567c27
9a8ad4567c27
1a97a9cc4d1b
1a97a9cc4d1b
10acc31def5d
10acc31def5d
f5f0b338bbd6
f5f0b338bbd6
```

可以输入下面的命令将这个镜像 ID 列表提供给 docker rmi 命令：

```
# docker rmi $(docker images -q)
Untagged: docker.io/ubuntu:trusty-20150320
Deleted: b7cf8f0d9e82c9d96bd7afd22...
Deleted: 2c014f14d3d95811df672ddae...
Deleted: a62a42e77c9c3626118dc4110...
Deleted: 706766fe101906a1a6628173c...
Untagged: docker.io/cricket/hangman:1.9
Untagged: docker.io/cricket/hangman:latest
Deleted: 1403322a81c5362762f84a26b...
Deleted: 834629358fe214f210b0ed606...
Deleted: 00a0c78eeb6d81442efcd1d7c...
Deleted: 511136ea3c5a64f264b78b543...
Error response from daemon: No such image: 1403322a81c5
Untagged: registry.access.redhat.com/rhel7/rhel-tools:7.1-9
Untagged: registry.access.redhat.com/rhel7/rhel-tools:latest
```

可以看到每个名字被取消标签以及每个镜像被删除。采用这种方式删除镜像可能导致多次请求删除同一个镜像名。如刚刚的例子所示，注意到有一个消息说 1403322a...开头的镜像不存在。如果向上查看一下，就会发现它已经被删除了，所以这不算是一个真正的错误。

除了用一个列表一次性地向 docker rmi 命令提供镜像，docker rmi 命令还容许在命令行上提供多个镜像名或镜像 ID。因此，删除多个镜像的另一种方法像下面这样：

```
# docker rmi 1403322a81c5 0114405f9ff1 53b6894a9c8a
Untagged: cricket/hangman:1.9
Untagged: cricket/hangman:latest
Deleted: 1403322a81c5362762f84a26b2c818fcc0c9f...
Untagged: centos:7
```

```
Untagged: centos:centos7
Untagged: centos:latest
Deleted: 0114405f9ff12fb7b012d0f7eb2f958c6ab86...
Deleted: 3d3c8202a57465ab6a24852559d21ca72a4af...
Deleted: b6718650e87e3706c52682c87ecfd7a7e1fc1...
Untagged: redis:latest
Deleted: 53b6894a9c8af40f9f12b722518b570cfb791...
Deleted: ef119c54808b9a791d89076b9d795d4cd157d...
Deleted: f94bd3b706e967987a9a433c1b3a909b7db60...
...
```

11.3 删除容器

每次运行容器，一个新的层会添加到容器中，实际上是创建了一个新容器。除非明确地指定要删除新容器（使用 docker run 的--rm 选项），否则它会继续保留在系统上，准备被重启或被保存为镜像。

删除容器比删除镜像要稍微复杂一些。要记住的最为重要的一点是，默认情况下，不能删除正在运行的容器。因此，要么需要先停止它，要么需要用强制选项（-f）来删除它。

与镜像一样，可以逐个删除容器或者向 docker rm 命令传递一个容器列表。在删除容器前，可以使用 docker ps 命令的-s 选项来检查每个容器消耗了多少空间。如果这样做，你也许会注意到容器实际上很小。这是因为容器的大小只是容器起始镜像加上额外部分的大小。

11.3.1 删除单个容器

要删除单个镜像，所要做的只是使用容器名或 ID 调用 docker rm 命令。接来下的例子展示了删除不同状态的容器的方法。

1. 删除正在运行的容器或暂停的容器

下面的例子是这个系统上运行的容器，一个已暂停而另一个正在运行。使用 docker ps 命令查看正在运行的容器：

```
# docker ps
CONTAINER ID  IMAGE                          COMMAND           CREATED
    STATUS                      PORTS      NAMES
705c1a4e30e0  rhel7/rhel-tools:latest "/usr/bin/bash"  9 seconds ago
    Up 51 seconds (Paused)                 rhel-tools
eaddcc59bc86  cricket/hangman:latest "go-fish"         4 hours ago
    Up 4 hours                             grave_pare
```

注意，尝试删除这些容器时，无论容器是正在运行还是暂停，命令都会以同样的方式失败：

```
# docker rm eaddcc59bc86
```

```
Error response from daemon: Conflict, You cannot remove a running
container. Stop the container before attempting removal or use -f
FATA[0000] Error: failed to remove one or more containers
# docker rm rhel-tools
Error response from daemon: Conflict, You cannot remove a running
container. Stop the container before attempting removal or use -f
FATA[0000] Error: failed to remove one or more containers
```

也要注意到，我在一个示例中使用了容器名，而另一个使用了容器 ID。无法删除暂停的容器，即便使用了强制选项。因此，我要取消暂停，停止它，而后再删除它。对于另一个容器，即便它尚在运行，我也可以用强制选项删除它：

```
# docker unpause myrhel-tools
myrhel-tools
# docker stop myrhel-tools
myrhel-tools
# docker rm myrhel-tools
myrhel-tools
# docker rm -f eaddcc59bc86
eaddcc59bc86
# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
#
```

如你所见，两个容器都不再运行，都已经被删除。

2. 删除已经停止的容器

容器停止后（假设没有通过--rm 选项运行它），新容器会留在系统上。可以再次启动它（使用 docker start），将其保存为镜像，或在不再需要它的时候删除它。输入下面的命令查看系统上运行过并已退出的容器：

```
# docker ps -a
CONTAINER ID  IMAGE          COMMAND                  CREATED
        STATUS                      PORTS     NAMES
d70cfe9cc624  ubuntu:latest  "/bin/bash"              16 seconds ago
        Exited (0) 9 seconds ago                 reverent_poitras
5f785798faae fedora:latest   "python -m SimpleHTT   2 minutes ago
        Exited (137) 4 seconds ago               MyWebServer
4b043f61e00   hangman:latest "go-fish"               6 hours ago
        Exited (0) 6 hours ago                   agitated_mcclintock
...
```

当删除容器时，使用容器 ID 或名字调用 docker rm 命令。下面是一些例子：

```
# docker rm reverent_poitras
reverent_poitras
# docker rm 5f785798faae
5f785798faae
```

11.3.2 删除多个容器

就像处理镜像时所做的那样，可以通过为 docker rm 命令提供一个容器名列表来删除多个容器。可以使用 docker ps -a -q 命令列出系统上所有已经不再运行的容器。例如：

```
# docker ps -a -q
16a6a35759cb
55a35f0736ac
04b043f61e00
8f94e788d173
9f43d1f6e699
176feb8ee722
0cd1ee87a368
cf1e1242b5a3
...
```

现在，要删除这些容器，只要像下面这样将 docker ps -a -q 命令的标准输出传给 docker rm 命令的标准输入：

```
# docker rm $(docker ps -a -q)
16a6a35759cb
55a35f0736ac
04b043f61e00
8f94e788d173
9f43d1f6e699
176feb8ee722
0cd1ee87a368
cf1e1242b5a3
...
```

尽管输出看起来一样（容器名的列表），但刚刚所示的命令会从系统中删除所有不再运行的本地容器。

11.4 清理和保存容器

如果向容器添加了一些软件或保存了一些数据，你也许会决定要用更为持久的方式保存这个容器，并将其存储为镜像。在第 5 章中，我描述了如何将容器保存并提交为镜像。这里，我会讨论在将容器保存为镜像前，你能做些什么来清理和缩减容器。

我为这个示例创建了两个容器：一个来自 fedora 基础镜像，一个来自 ubuntu 基础镜像。这两个 Linux 系统用不同的工具管理软件包，所以需要用不同的方法着手清理这些容器。



注意

尽管已经有可用的工具将容器保存为镜像，但这不一定是创建你所需镜像的最有效的方法。更好的方法是使用 docker build 从基础镜像构建你想要的容器，其中包括你认为合适的任何清理命令。第 12 章中会描述 docker build 命令。

11.4.1 清理和保存 Ubuntu 容器

我在一个 ubuntu 基础镜像上执行了 docker run，打开 shell 并用 apt-get install 安装了几个服务器软件包。在停止容器后，我再次连接上它，并运行下面的命令对其进行清理和缩减其大小：

```
# docker start -ai 369f25938d44
root@369f25938d44:/# apt-get remove unneeded_packages
root@369f25938d44:/# apt-get clean
root@369f25938d44:/# rm unneeded_files
root@369f25938d44:/# exit
# docker export 369f25938d44 > myubuntu.tar
# docker rm 369f25938d44
# cat myubuntu.tar | docker import - mynewubuntu:latest
699c4def6a4cc14e75417bdf5c9d2e1769745845fd905cb989522bc166cafa3c
# docker images
REPOSITORY    TAG       IMAGE ID       CREATED              VIRTUAL SIZE
mynewubuntu   latest    699c4def6a4c   About a minute ago   257.4 MB
```

apt-get remove 和 rm 命令只是提醒你删除你觉得不再需要的软件包和文件。在退出容器运行的 shell 之后，我将容器导出为一个名叫 myubuntu.tar 的 tarball 文件。之后，我可以删除旧容器，因为我不再需要它了。

接下来，我用管道将 tarball 的内容传递给 docker import 命令，这会把它保存成本地系统的一个镜像，这个镜像的名字是 mynewubuntu:latest。这之后，该镜像会出现在 docker images 命令显示的列表中并且已经准备好可供使用了。

11.4.2 清理和保存 Fedora 容器

这次从 fedora 基础镜像开始，我执行 docker run 打开了一个 shell 并使用 yum install 安装了几个服务器软件包。在停止容器后，我再次连接上它，并运行下面的命令对其进行清理和缩减其大小：

```
# docker start -ai 6d64eb633e02
bash-4.3# yum remove unneeded_packages
bash-4.3# rm unneeded_files
bash-4.3# yum clean all
```

```
Cleaning repos: fedora updates
Cleaning up everything
bash-4.3# exit
# docker export 6d64eb633e02 > myfedora.tar
# docker rm 6d64eb633e02
# cat myfedora.tar | docker import - mynewfedora:latest
7a0b47e65b1f028bec94420a234c45beb48f02c0a8d7c222ac999a5913b16d3b
# docker images
REPOSITORY    TAG      IMAGE ID      CREATED                VIRTUAL SIZE
mynewfedora   latest   7a0b47e65b1f  About a minute ago     245.7 MB
```

与之前的 ubuntu 容器示例一样，我删除了不需要的软件包和文件，清理了安装软件包所遗留的元数据，而后退出 shell。随着容器停止——但它仍在系统上，我将容器导出为一个 tarball 并将旧容器删掉。为了将 tarball 再次转换成系统内的镜像，我将 myfedora.tar 文件的内容传送到标准输出（使用 cat 命令），并使用管道将其传递给 docker import 命令，并将新镜像命名为 mynewfedora:latest。

11.5 小结

在镜像和容器被拉取和运行后，Docker 会将它们保存起来。如果不小心的话，它会在用户没有察觉的情况下消耗系统的大量存储空间。通常而言，要密切关注/var/lib/docker 目录中的可用磁盘空间，因为这是 Docker 保存镜像和容器的地方。

应该每隔一段时间检查一下系统上存储的镜像和容器并清理掉不再需要的容器。运行 docker images 来查看系统上存储的镜像并用 docker rmi 删除它们。对于容器，运行 docker ps 和 docker ps -a 来查看系统内的容器并用 docker rm 删除它们。

要更为持久地保存容器的内容，可以使用 docker export 和 docker import 将容器保存为一个 tarball，然后再将其转换成系统内的镜像。然而，在这样做之前，要把容器清理一下——删除不需要的缓存数据、软件包和文件——是个不错的主意。

第 12 章

构建容器镜像

本章主要内容

- 构建一个简单的 Docker 容器镜像
- 使用 Dockerfile 指令
- 学习一些技巧让镜像构建更为高效

尽管可以通过向正在运行的容器添加软件并将其提交成镜像（docker commit）的方式来创建 Docker 镜像，但是创建 Docker 镜像的首选方法是使用 docker build 命令。要创建一个文件来保存构建 Docker 镜像的指令，这个文件被称为 Dockerfile。

鉴于以下这些原因，由 Dockerfile 构建 Docker 镜像要优于手工修改和提交。

- **可重建**：如果向正在运行的容器添加软件并将其提交为镜像，那么创建该镜像所做的事情并没有被确切地记录。有了 Dockerfile，就可以确切地看到向基础镜像添加了什么，并且只要通过这个文件再次构建该镜像就可以重建它。
- **可修正**：如果构建该镜像时有错误，只要修改 Dockerfile 并再次运行 docker build 就可以了。实际上，由于 docker build 会为每一步创建一个新容器层，因此稍后的构建会从最近成功的那一步开始。如果只是靠近构建结束的一两步需要改正，这就能在构建过程上节约大量时间。
- **可移植**：虽然 Docker 镜像可能比较小，但与之相比，即便是一个巨大的 Dockerfile，其尺寸也是微不足道的。如果你把 Dockerfile 交给某些能够访问构建所用的基础镜像（它可能在公共 registry 中）以及构建中所需的其他文件的人，他们就能重建你的容器镜像并在他们的系统上重用它，而不必传输整个镜像。
- **可验证**：只要有一个已知的基础镜像和一个 Dockerfile，人们就能够验证有哪些东西被拉取到他们构建的容器中。如果有人只交给你一个完成的容器，你将不得不打开它并四处查看以了解它中间有什么东西。
- **可更新**：如果 Dockerfile 使用 yum 命令（或 dnf）或 apt-get 命令来安装软件包，除非你对这些软件包有明确的版本要求，否则构建会获取你要求的软件包的最新版本。因此，一次简单的重新构建所产生的 Docker 镜像会产生一个包含来自你配置

的 repository 的最新软件。

本章将逐步指导读者从 Dockerfile 构建自己的镜像。在这个过程中，我描述了有效使用 Docker 进行镜像构建的技巧——这些镜像会被高效地存储和运行。

一旦了解了构建 Docker 镜像的过程和组件，就可以参考第 18 章中的一些复杂而有趣的 Dockerfile 示例，以此从能构建的各种 Docker 镜像中获取灵感。

12.1 进行简单的 Docker 构建

假设 Docker 服务在你的系统上已经运行起来了，要构建自己的 Docker 镜像，你所需的只是一个 Dockerfile、访问想作为起始的基础镜像，以及想要添加到镜像中的其他任何文件。对于 docker build 命令，需要做的是指明构建所依赖的 Dockerfile 文件在哪里，这通常会用一个点（.）来表明当前目录。

还可以为 docker build 设置其他选项，稍后会介绍。现在，下面的步骤用来创建你的第一个简单的 Docker 镜像。

（1）**创建存放 Dockerfile 的目录**。创建一个目录来保存你的 Dockerfile。例如：

```
# mkdir ~/Imagedir
# cd ~/Imagedir
```

（2）**创建 Dockerfile**。使用任何文本编辑器，创建一个 Dockerfile 文件，其内容如下（将 MAINTAINER 行替换为你的名字）：

```
# Character-based games container image
FROM fedora:latest
MAINTAINER John W. Jones
RUN dnf install -y bsd-games words
# Start the application
ENTRYPOINT ["/usr/bin/hangman"]
```

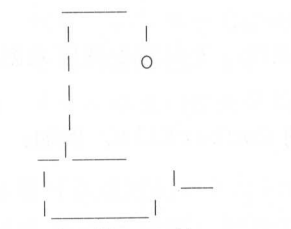
（3）**构建镜像**。输入下面的命令（Dockerfile 所在的当前目录）来构建镜像。-t 选项用来将镜像命名为 myhangman：

```
# docker build -t myhangman .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM fedora:latest
---> ded7cd95e059
Step 1 : MAINTAINER John W. Jones
---> Running in 7b56400206f3
---> 58f0c53fc128
Removing intermediate container 7b56400206f3
Step 2 : RUN dnf install -y bsd-games words
---> Running in 706da0a49bdf
...
```

```
Installed:
  bsd-games.x86_64 2.17-46.fc22                words.noarch 3.0-23.fc21
Complete!
---> c889372d5ed1
Removing intermediate container 706da0a49bdf
Step 3 : ENTRYPOINT /usr/bin/hangman
---> Running in 9a9bcb54cee4
---> 934e99e33296
Removing intermediate container 9a9bcb54cee4
Successfully built 934e99e33296
```

（4）**运行镜像**。使用 `docker run` 命令运行镜像。这会在当前 shell 中启动 hangman 游戏。`-it` 选项交互式地运行容器：

```
# docker run -it myhangman
```



```
Guessed: acdeiorstu
Word #:          3
Current Average: 4.667
Overall Average: 6.500
 Word: directdiscourse
Guess:
You got it!
Another word? n
```

（5）**检查镜像**。使用下面的命令查看新镜像以及前面运行的容器已在你的系统上：

```
# docker images
REPOSITORY          TAG     IMAGE ID      CREATED         VIRTUAL SIZE
myhangman           latest  934e99e33296  9 minutes ago   319.9 MB
docker.io/fedora    latest  ded7cd95e059  5 weeks ago     186.5 MB
# docker ps -a
CONTAINER ID  IMAGE             COMMAND                  CREATED
   STATUS                     PORTS    NAMES
6a507de124ff  myhangman:latest "/usr/bin/hangman"  9 minutes ago
   Exited (0) 6 minutes ago           jolly_darwin
```

如果一切顺利，你应该可以玩来自容器的 hangman 游戏了。以下几点有助于理解构建过程中发生的事情。

- **Docker 守护进程执行构建**：`docker build` 命令实际上并未构建容器。这个命令将构建所需的信息传递给 Docker 守护进程。
- **限制构建目录**：存放 `Dockerfile` 文件的目录只包含构建所需的文件。这是因为当前目录或其子目录中的任何文件都会被发送给 Docker 守护进程并且可供 ADD 命令添加到容器中使用。因此，如果把 `Dockerfile` 放在`/var` 目录中，`/var` 目录下的每一个文件和目录都会被发送给 Docker 守护进程，以便可能会包含到容器中。这将

大大减慢构建过程并消耗过多不必要的磁盘空间。

- **中间容器**：每个指令会向现存镜像添加一个新的层，并提交生成一个中间容器镜像。使用这种方法，每次在相同的 Dockerfile 上运行 docker build 时，构建可以从最近成功的指令开始并从那里继续。
- **构建选项**：构建镜像时至少要使用-t 选项为该镜像分配一个名字。如果计划将该镜像推送到 registry 中，可以包含用户名（如 cricket/myhangman）或主机名（如 registry.example.com:5000/myhangman）。其他构建选项将在本章稍后介绍。
- **基础镜像（FROM）**：如果系统已连上因特网，会从 Docker Hub Registry 拉取 fedora:latest 基础镜像（位于 FROM 行）。如果该镜像已被拉取，会从本地系统使用该镜像。除了注释，FROM 必须是 Dockerfile 的第一个指令。:latest 标签实际上并不是必需的，因为如果没有输入标签，会假定标签是:latest。但是，如果想使用一个特定版本的基础镜像，一定要添加该标签（如 fedora:22）。
- **MAINTAINER**：这表明了镜像的作者。这个信息会被保存在容器内。通常要添加你的全名和邮件地址。
- **RUN**：放置在 RUN 行中的命令会在构建过程中（不是之后容器自己运行时）运行。fedora 基础镜像能够从 Fedora 的 repository 中安装软件包（使用 yum 或 dnf 安装命令）。在这个例子中，我通过一个 RUN 行安装了 bsd-games 和 words 软件包。通过 Dockerfile，我能够使用 RUN 指令运行我想要的任何命令，在构建时改变镜像。
- **ENTRYPOINT**：这一行指明当使用 docker run 启动容器时要执行的命令——在本例中是/usr/bin/hangman。ENTRYPOINT 基本上表明了容器镜像的意图。当运行容器时，可以使用--entrypoint=""覆盖 ENTRYPOINT 命令。然而，让人在运行镜像时覆盖默认命令的更简单的方法是使用 CMD，而不是 ENTRYPOINT（稍后描述）。

这个示例用来说明构建一个简单容器的基本过程。然而，还有很多指令可以加到 Dockerfile 中来定制你的容器。许多指令将在本章接下来的几节中介绍。

12.2 通过 Dockerfile 设置要执行的命令

CMD 和 ENTRYPOINT 指令确定启动容器时要运行的命令。然而，RUN 指令设置的命令仅在构建镜像时会被使用，而 CMD 和 ENTRYPOINT 会在执行 docker run 时被使用。如接下来的 Dockerfile 示例中所说明的，这 3 个指令有不同的意义。

12.2.1 使用 CMD 指令

通过CMD，可以确定镜像要运行的默认命令，以及想传递给它的选项。如果 Dockerfile

中没有 ENTRYPOINT，当启动容器镜像时默认运行的就是 CMD 的值所指定的命令。如果 Dockerfile 中有 ENTRYPOINT，ENTRYPOINT 的值会是要运行的命令，而 CMD 的值会被当作 ENTRYPOINT 的选项。

在 Dockerfile 中可以只有一个 CMD 行。下面是一个 Dockerfile 示例，它包含了一个 CMD 指令，该指令运行 cat 命令展示了几个文件：

```
FROM fedora:latest
MAINTAINER John Jones
CMD ["cat","/etc/hosts","/etc/os-release"]
```

在这个例子中，如果你构建了一个名为 mycmd 的镜像，docker run mycmd 会将 hosts 和 os-release 文件输出到 shell 上。这个例子中的 CMD 参数是一个命令（cat）和紧随其后的两个文件名（/etc/hosts 和/etc/ os-release）。这些参数是 JSON 格式的（方括号以及逗号分隔的双引号列表）。这行 CMD 可以只是 CMD cat/etc/hosts /etc/os-release，但这会让命令被一个 shell 执行（sh -c），而不是直接执行。

当运行镜像时，可以覆盖 CMD 指令。因此，注意用两种不同的方法运行 mycmd 的不同结果：

```
# docker build -t mycmd .
# docker run mycmd                  运行 cat 展示 hosts 和 os-release 文件
# docker run -it mycmd bash         运行 bash，忽略 cat 命令
```

只要在 docker run 命令的末尾添加参数，容器内的 CMD 指令就会被忽略。因此，第二个例子打开一个 bash shell，而不是运行 cat 命令。如果想指定一个不会被 docker run 命令的结尾选项覆盖的命令，那就使用 ENTRYPOINT 指令。

12.2.2 使用 ENTRYPOINT 指令

使用 ENTRYPOINT 指令可以定义运行容器镜像时所要执行的命令。它的实现方式使你无法通过在 docker run 命令行结尾添加参数来进行覆盖。如果 Dockerfile 包含 ENTRYPOINT 指令并且还有 CMD 指令，CMD 指令行的参数会传递给 ENTRYPOINT 行定义的命令。下面是使用了 ENTRYPOINT 指令的 Dockerfile：

```
FROM fedora:latest
MAINTAINER John Jones
CMD ["/etc/hosts","/etc/os-release"]
ENTRYPOINT ["cat"]
```

由于该 Dockerfile 使用 ENTRYPOINT 设置运行 cat 命令，容器默认会使用该命令。然而，该文件中还有 CMD 行，这一行的参数会被传递给 ENTRYPOINT 指令。如果你已经将这个 Dockerfile 构建成名为 myent 的镜像，下面是运行结果镜像的一些例子：

```
# docker build -t myent .
```

```
# docker run myent                          使用 hosts 和 os-release 运行 cat
# docker run -it myent /etc/issue           使用 issue 运行 cat，忽略其他参数
```

注意，只要没有为 docker run 命令提供其他参数，ENTRYPOINT 行的命令（cat）就会将/etc/hosts 和/etc/os-release 文件作为参数。但是，如果运行镜像时提供了参数（这里是/etc/issue），CMD 指令的内容就会被忽略。

只用 ENTRYPOINT 指定命令的好处是，你可能想覆盖命令运行的默认方式。考虑下面的例子：

```
FROM fedora:latest
MAINTAINER John Jones
ENTRYPOINT ["cat","/etc/hosts","/etc/os-release"]
```

构建和运行这个镜像时，会得到如下结果：

```
# docker build -t myent1 .
# docker run myent1                         使用 hosts 和 os-release 运行 cat
# docker run -it myent1 /issue              使用全部 3 个文件运行 cat
```

由于 hosts 和 os-release 文件位于 ENTRYPOINT 行中，因此在 docker run 命令行结尾添加参数并不能覆盖它们。在 docker run 命令行结尾添加/etc/issue 会在显示其他两个文件之后显示该文件。

关于 ENTRYPOINT 要牢记最后一件事。即使是 docker run 命令行的结尾参数也不能覆盖 ENTRYPOINT 设置的命令，但使用--entrypoint=""选项可以覆盖 ENTRYPOINT 设置的命令。例如，下面的命令行将运行 bash shell，而不是运行所选镜像内由 ENTRYPOINT 设置的默认 cat 命令：

```
# docker run -it --entrypoint="/bin/bash" myent1
```

12.2.3 使用 RUN 指令

如同之前提到的，RUN 指令只在使用 docker build 创建镜像时才会被解释和执行。RUN 指令的目的通常是执行那些以某种方式改变镜像的命令。例如，你可以安装软件包或创建会成为这个镜像的一部分的配置文件。

在这个例子中，我在构建时创建了一个文件，并让之后的 docker run 来显示它：

```
FROM fedora:latest
MAINTAINER John Jones
RUN echo "This container was built on $(date)." > /tmp/built.txt
ENTRYPOINT ["cat","/tmp/built.txt"]
```

当我运行 docker build 时，该命令会读取当前日期和时间，将其插入一个句子中并将这个句子输出到/tmp/build.txt 文件中。因为是在构建时运行的 echo 命令，所以每次我运行 docker run 命令时都会显示相同的日期：

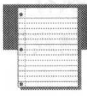
```
# docker build -t myrun .
# docker run myrun                                    查看镜像构建的时间
This container was built on Sun Jul 5 13:47:51 EDT 2015.
# docker run myrun                                    注意构建时间是一样的
This container was built on Sun Jul 5 13:47:51 EDT 2015.
```

12.2.4 通过 Dockerfile 向镜像添加文件

可以使用 ADD 指令在构建时向容器添加所选的文件。当使用 ADD 向镜像添加文件和目录时，docker build 会将宿主机系统内包含 Dockerfile 的目录作为根目录（/）和当前目录。在下面这个例子中，我在 Dockerfile 所在目录中创建了一个文件并将其添加到名为 myadd 的镜像中，具体如下：

```
# mkdir ~/Myadd
# cd ~/Myadd
# echo "This is the test.txt file." > test.txt
# vi Dockerfile
# cat Dockerfile
FROM fedora:latest
MAINTAINER John Jones
ADD /test.txt /tmp/test.txt
ENTRYPOINT ["cat","/tmp/test.txt"]
# docker build -t myadd .
# docker run myadd                                    查看 test.txt 文件的内容
This is the test.txt file.
```

在 ADD 指令行中，/test.txt 在宿主机系统上与 Dockerfile 位于同一目录，而 /tmp/test.txt 显示的是该文件去往容器内的位置，我可以仅仅将宿主机的文件标识为 text.txt，因为 docker build 会将 Dockerfile 所在目录作为当前目录并会查看该目录中的文件。再说一次，记得要把所有要构建进容器的文件或目录放在 Dockerfile 所在的目录或子目录中。



注意

如果你常常重新运行 docker build，只要一个指令运行成功，新的 docker build 就会重用该指令的缓存结果。ADD 和 COPY 指令是一个例外，如果底层文件发生了变化，它们将会重新添加或复制该文件。参考 12.7.4 节，了解该问题更为全面的描述。

12.3 通过 Dockerfile 暴露镜像的端口

在 Dockerfile 中添加 EXPOSE 指令可以从所构建的镜像暴露特定的端口。当一个端口从正在运行的容器镜像中暴露出来时，它使以下两件事成为可能。

- **链接容器**：一旦运行镜像，只要将正在运行的容器链接到另一个容器，其他容器就可以使用被暴露的端口，就像在同一本地系统上可用一样。
- **运行时暴露**：镜像构建时由 EXPOSE 指令标识的任何端口都很容易通过本地宿主机的相同端口暴露出来。对 docker run 使用-P 选项，所有从容器内暴露的端口都被随机分配给宿主机上的端口。之后可以在结果容器上运行 docker port 命令，查看端口的映射情况。

下面是一个容器镜像的 Web 服务器 Dockerfile，该镜像暴露了 80 端口：

```
FROM fedora:latest
MAINTAINER John Jones
RUN yum install -y httpd
EXPOSE 80
# Start the service
CMD ["-D", "FOREGROUND"]
ENTRYPOINT ["/usr/sbin/httpd"]
```

当要运行该镜像时，下面是借助 docker run 命令将容器的 80 端口暴露给本地系统的 8080 端口的方法：

```
# docker run -p 8080:80 -d myweb
# curl localhost:8080 | head
```

-p 8080:80 表明宿主机的 8080 端口指向容器内的 80 端口。curl 命令行会显示 Apache HTTP Server Test 页面的前几行。

如果没有为端口号指定协议，默认假定是 TCP 协议。要指定不同的协议，如 UDP 协议，可以在端口号后面紧跟一个斜线（/）和协议名。例如，可以用 53/udp 开放 UDP 的 53 端口。

12.4 在 Dockerfile 中设置环境变量

使用 ENV 指令，可以为所选的任何键设置环境变量。接下来继续处理 Dockerfile 的后续指令可以使用这些变量。可以在一行上设置多个变量，也可以使用多个 ENV 行。

Dockerfile 中的环境变量对于这类事情非常有用：修改数据目录的位置，添加密码或设置用户名。WordPress 的官方镜像（https://hub.docker.com/_/wordpress/）设置了多个环境变量。可以修改 Dockerfile 中的环境变量（ENV *variable key*）或通过 docker run 命令行在运行时修改它们（-e *variable=key*）。

例如，可以将下面的环境变量构建到 Dockerfile 中来创建 WordPress 镜像：

```
ENV WORDPRESS_DB_HOST=host01.example.com
ENV WORDPRESS_DB_USER=root
ENV WORDPRESS_DB_PASSWORD=MYrut3pZwrd
```

当运行这个 WordPress 镜像时，可以通过 docker run 命令的-e 选项修改这些变量。Docker Hub Registry 上的 WordPress repository 就如何修改这些变量提供了下面这个例子：

```
# docker run --name some-wordpress -e WORDPRESS_DB_HOST=10.1.2.3:3306 \
    -e WORDPRESS_DB_USER=... -e WORDPRESS_DB_PASSWORD=... -d wordpress
```

12.5 在 Dockerfile 中设置标签

可以在 Dockerfile 中使用 LABEL 指令给所选的键设置值。你可以用想用的任何键名，因为在如何设置这些标签方面没有限制。正因如此，Docker Project 就使用 LABEL 指令提出了一些建议。

下面是使用 LABEL 指令的一个简单例子：

```
LABEL description="Simple web server container"
```

如果将这个 LABEL 构建到名为 myweb 的镜像上，可以按如下方式使用 docker inspect 查看这个标签的设置方式：

```
# docker inspect myweb | less
...
        "Hostname": "26bc48f09da7",
        "Image": "6396baf5ba25e16...",
        "Labels": {
            "description": "Simple web server container"
        },
...
```

下面是在 Dockerfile 中使用 LABEL 指令的一些技巧。

- **一行多个标签**：如果需要为镜像设置多个标签，用一个 LABEL 行设置可以防止为每个标签创建一个单独的容器层。下面是在一个 LABEL 行设置多个标签的两种不同方法：

```
LABEL description="My game image" department="Sales" user="joe"
LABEL description="My game image" \
    department="Sales" \
    user="joe"
```

- **使用唯一标签**：如果在 Dockerfile 中多次设置一个标签，最后一个值会被使用。Docker Project 使用标签时，它在每个标签前附加了 com.docker.*、io.docker.*或 org.dockerproject.*。对你而言，要在创建标签时保证避免标签冲突，用受你控制的域名的反向 DNS 表示法是一个好办法，如 net.linuxtoys.mylabel。
- **标签的字符**：标签中可以使用的字符有一些限制。尽管不是强制，但应该在键名中只

使用字母（英文字母）、数字（阿拉伯数字）、点和中划线。确保每个键以字母或数字开头和结尾。不要使用连续的点或中划线。

- **用标签过滤**：镜像构建后，可以使用--filter 选项列出那些只包含你设定标签的镜像或容器。例如：

```
# docker ps -a --filter "label=description=My game container"
CONTAINER ID  IMAGE                          COMMAND     CREATED
   STATUS                        PORTS                   NAMES
c35fe57d078c  cricket/hangman:latest      "hangman"    4 minutes ago
   Exited (0) 4 minutes ago
# docker images --filter "label=description=My game container"
REPOSITORY       TAG      IMAGE ID       CREATED          VIRTUAL SIZE
cricket/hangman  latest   1403322a81c5  10 minutes ago   427.3 MB
```

既然已经对 Dockerfile 中可以使用的许多指令有所了解，接下来一节描述将 Dockerfile 构建为镜像时，能用于 docker build 命令的一些其他选项。

12.6 使用其他 docker build 命令选项

尽管将 Dockerfile 构建为 Docker 镜像唯一需要的就是 docker build，但还是可以向 docker build 添加其他选项来调整构建过程。下面这些 docker build 命令行用以说明不同的选项：

```
# docker build --force-rm=true .
# docker build --no-cache=true .
# docker build -f ~/Myweb/Dockerfile01 ~
# docker build --pull=true .
```

--force-rm=true 选项告诉 Docker 强制删除所有中间镜像，即使某些时候构建失败了。然而，删除中间容器并不会删除缓存数据。因此，要让构建真正从头开始，使用 --no-cache=true 选项。通过设置--no-cache=true，docker build 在构建中不会使用任何缓存数据。每一步都从零开始。

默认情况下，当明确了构建目录——通常是表示当前目录的一个点（.），会将该目录中名为 Dockerfile 的文件用于构建。通过添加-f 选项，告诉 docker build 使用一个不同的文件来构建该镜像。记住，指明的构建文件必须在构建目录中或者该目录的子目录中。在刚刚展示的例子中，构建目录是用户的主目录（~），而构建文件名为 Dockerfile01，该文件位于构建目录的子目录中（-f ~/Myweb/ Dockerfile01）。

如果所需的基础镜像（FROM 行）之前已经被 Docker 拉取到本地系统，默认情况下 docker build 不会再查找该镜像的新版本。通过对 docker build 添加--pull=true，Docker 会检查是否可以从启用的 repository 中找到该镜像的新版本。

12.7 构建容器的技巧

如果你准备好开始构建容器，这里有一些技巧可以帮你让构建过程更好、更方便。

12.7.1 清理镜像

要让镜像尽可能小，要将不需要的软件包和数据从镜像中清除掉。如果是用 yum 或 apt-get 安装的软件包，要清除掉该过程产生的任何缓存数据（如 RUN yum clean -y all）。类似地，可以从镜像中移除不需要的软件包（如 RUN yum remove -y vim-minimal）。

12.7.2 保持构建目录较小

像之前提到的，在构建过程开始前，构建目录中的每个文件都会复制给 Docker 守护进程并被存储（默认是在/var/lib/docker 中）。开始构建前，删除该目录及其子目录中的文件，加速构建并节省磁盘空间。

在某些情况下，从构建目录结构中删除文件可能不太方便，可以将这些文件排除在构建之外。作为替代方案，要从构建目录结构排除文件，可以向构建目录添加一个.dockerignore 文件并在其中记录要忽略的文件。下面是可以在.dockerignore 文件中使用的各种记录：

```
passwd
passwd*
*/passwd*
*/*passwd*
passw?
!passwd-local
mystuff/
```

下面解释了这些例子如何在构建过程中防止将文件或目录从构建目录复制到 Docker 守护进程。

- **passwd**：只忽略构建目录中的 passwd 文件。
- **Passwd**：构建目录中所有以 passwd 开头的文件都会被跳过。例如，passwd-stuff、passwd.5.gz 和 passwd.html 都会被忽略。但任何子目录中的 passwd 都不会被忽略。
- **/passwd**：构建目录的直接子目录中，所有以 passwd 开头的文件都会被忽略。
- **passw?**：构建目录中任何名字以 passw 开头并且还有额外一个字母的文件都会被忽略，如 passwd、passwa、passwb 等。
- **!passwd-local**：如果该条目之前的任何规则造成名为 passwd-local 的文件被忽略，这里所示的字符串会让 passwd-local 文件不再被忽略。如果引起

passwd-local 被忽略的规则在这条规则之后出现，passwd-local 文件会被忽略。

- **mystuff/**：如果构建目录中有名为 mystuff 的目录，忽略该目录以及其下的任何文件和子目录。

12.7.3 保持容器简单

创建只做一件事情的容器（运行文件服务器，启动 Web 浏览器，或提供一个简单的数据库）会简化之后对容器使用的管理和扩展。通常而言，每个容器应该只运行一个进程。如果一个容器需要来自其他容器的服务，可以将容器链接到一起或使用编排服务（如 Kubernetes）将一个容器需要的服务与提供这些服务的容器关联起来。

12.7.4 管理进行缓存的方法

每次从 Dockerfile 重新构建镜像，Docker 都会检查当前指令是否已经成功运行，从而在缓存中保存该指令的结果。如果结果是成功的并且被缓存了，默认情况下，Docker 会使用该指令的缓存数据并在新的构建中重用它。例如，如果指令所做的是下载并安装 100 个软件包这样的工作，这会节省大量的时间。

有时候使用缓存是正确的做法。而其他时候，可能要使用--no-cache=true 选项。例如，你了解到 docker build 过程中安装的软件包有了新的版本。如果你使用完全一样的指令再次构建这个镜像，Docker 并不知道要重新运行指令来获取新的软件包。避免使用之前缓存的一个办法是对指令进行某种修改。这会使指令重新运行，而不使用缓存。

如果构建中使用了 ADD 或 COPY 指令添加文件，当对相同的 Dockerfile 再次运行 docker build 时，对这些文件的任何修改都会被注意到。这是由于 Docker 会对 ADD 或 COPY 指令拉取的每个文件运行校验和（checksum），如果文件自身的内容或与文件关联的元数据（权限、日期/时间戳等）发生了改变，新文件会被复制给镜像。

刚刚提到的这些技巧应该为构建最初的容器提供了一个良好的开端。要了解构建容器方面更高级的建议，请参考第 17 章。

12.8 小结

docker build 命令是创建 Docker 容器镜像的首选方法。与修改运行容器并将其提交为新镜像相比，使用 docker build 构建镜像有诸多优点。通过 docker build 创建新镜像，可以更为容易地验证其内容、重建镜像，以及拉取最新的软件和配置信息。

用来构建 Docker 容器镜像的文件被称为 Dockerfile。你创建 Dockerfile 来确定构建所依赖的基础镜像、构建过程中要运行来修改容器的命令，在容器镜像被运行时设置要执行的命令，以及将其保存为你命名的新镜像。你还可以向 Dockerfile 添加其他指令来完成

下面这些事情：设置环境变量，从构建目录拉取文件，以及标识用户账号——该账号会运行你包含在 Dockerfile 中的命令。

本章结束了本书的第二部分。至此，我已展示了如何使用单独的镜像和容器。读者应该已了解如何运行、检视、罗列、启动、停止、链接以及处理单独的容器和镜像。本书接下来的部分将通过运行 Project Atomic 这类面向容器的操作系统，把容器带入云环境中。

第三部分

在云环境上运行容器

第 13 章

使用超级特权容器

本章主要内容

- 了解超级特权容器
- 使用 atomic 命令管理容器
- 使用超级特权容器

那些为运行容器而构建的操作系统都意在精简。与传统操作系统相比，Atomic Host 和 CoreOS 这类面向容器的操作系统通过仅提供一个最小功能集来获得更快的启动时间，以及更小的存储需求和更少的 CPU 消耗。这使其成为用于云环境部署的优秀的操作系统。

这些小而高效的系统的潜在缺点是，它们没有内置全功能操作系统所具有的全部工具。因此，用户需要学习新方法来管理这些系统。

本章关注如何使用所谓的超级特权容器（super privileged container，SPC）来添加软件工具，以便在 Atomic Host 系统上开展工作。通过 SPC，既能够获得需要的工具来管理 Atomic Host 系统，还能够以一种使这些工具能够摆脱其容器而作用于宿主机的方式来运行 Atomic Host 系统。

尽管本章描述的 SPC 能够用在常规的 Fedora、CentOS、Red Hat Enterprise Linux（RHEL）和 Atomic Host 系统上，但 SPC 在 Atomic Host 上最有用。原因是常规 RHEL 或 Fedora 系统很容易就能添加软件——只要使用 `yum`（或 `dnf`）和 `rpm` 命令。由于 RHEL Atomic Host 的特性（使用 Atomic 升级来添加软件）不允许使用 `yum` 或 `rpm` 向系统添加软件包，所以需要作用于 Atomic Host 系统的所有特定工具必须通过添加包含这些工具的容器来添加。本章所述的 SPC 是为向 Atomic Host 系统添加那些 Atomic Host 系统上不存在的工具而专门创建的。



注意

尽管 `docker run` 命令的特定选项能够启动 SPC，但本章重在讨论容器如何包含预设的 `docker run` 选项——可以利用这些选项在 Atomic Host 系统上使用 `atomic` 命令。

13.1 在 Atomic Host 中使用超级特权容器

Project Atomic 系统使用 atomic 式的升级来添加软件，而不是使用 rpm 和 yum 命令这样的传统工具来安装软件包。尽管 Atomic Host 系统是由 RPM 软件包构建的，但 Atomic Host 构建完成之后就不能向 Atomic Host 添加 RPM 软件包或从中删除软件包了。

因此，当 Atomic Host 系统需要更多软件时，最好的办法是将软件放入容器并在宿主机上运行这个容器。因为默认情况下容器不允许作用于宿主机本身，所以这些类型的容器需要额外的特权。

在 Atomic Host 系统中，SPC 被授予了特殊的权限来洞悉宿主机系统并在需要时改变宿主机系统。一个普通容器只能看到容器中运行的进程，而 SPC 能够看到并可能作用于宿主机上运行的所有进程。类似地，运行自 SPC 的命令可以直接访问宿主机系统上的设备并能查看宿主机的网络接口。

13.2 了解超级特权容器

从容器内运行一个应用程序时，通常会让该应用程序的处理与宿主机系统上其他容器保持分离，也会与宿主机本身保持分离。这是通过为容器维护受限的权限和独立的命名空间来做到的。SPC 是专为突破这些限制而设计的。

可以使用 `docker run` 命令的命令行选项来告知所选的容器镜像使用宿主机的命名空间，而不是容器的命名空间。`atomic` 命令是专为读取这些包含在容器元数据中的选项并将其传递给 `docker run` 命令而设计的。因此，理解 SPC 工作原理的前提是了解提供这些功能的 `docker run` 选项。

接下来一节会描述那些能够通过 `docker run` 命令行选项开放给 SPC 的宿主机权限和命名空间。

13.2.1 开放给宿主机的权限

正在运行的容器通常不能直接访问宿主机上的设备。同样地，当容器（默认情况下）运行在提供了 SELinux 的系统（如 Fedora 和 RHEL）中时，其访问宿主机上其他文件和进程的能力会由于其运行于受限的 SELinux 安全上下文中而被限制。

当 `docker run` 使用`--privileged` 选项时，容器几乎可以像容器外运行的进程一样访问宿主机系统。然而，为了能够使用这些权限，不得不开放宿主机上不同的命名空间，以便它们能在该容器中被看到。

13.2.2 访问宿主机的进程表

默认情况下，正在运行的容器仅能看到自己的进程表。因此，对容器而言，进程 ID（PID）

为 1 代表了由容器启动的进程，而不是在通常的（运行中的）Linux 系统上看到的 systemd 或 init 进程。容器进程表上的其他进程是那些在容器内启动的进程。

运行容器时，如果不告知 docker run 要使用宿主机的进程表，就会限制容器的视角，不让其看到宿主机正在运行的进程。例如，如果用 docker run 启动 bash shell，而后在该容器内运行 ps 命令查看正在运行的进程，所看到的内容大概像下面这样：

```
# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  2 16:06 ?        00:00:00 /bin/bash
root           7       1  0 16:06 ?        00:00:00 ps -ef
```

注意，进程 ID 为 1 的就是 bash shell，而 ps 命令的 PID 是 7。如果运行该容器时我提供了--pid=host 选项来设置该容器使用宿主机的进程表而不是容器自己的进程表，容器内运行的 ps 可能看起来像下面这样：

```
# ps -ef
UID    PID    PPID  C STIME TTY          TIME CMD
root     1       0  0 Apr24 ?    00:00:09 /usr/lib/systemd/systemd...
root     2       0  0 Apr24 ?    00:00:00 [kthreadd]
root     3       2  0 Apr24 ?    00:00:14 [ksoftirqd/0]
root     5       2  0 Apr24 ?    00:00:00 [kworker/0:0H]
...
root 18941   18864  0 17:02 ?    00:00:00 ps -ef
```

ps 命令输出的列表显示了宿主机上运行的所有进程。

13.2.3 访问宿主机的网络接口

默认情况下，Docker 守护进程会通过 DHCP 为宿主机 docker0 网络上的每个容器提供一个 IP 地址，该地址会分配给 eth0 接口。在 Fedora 和 RHEL 系统中，docker0 默认使用的 IP 地址范围是 172.17.0.0/16。

使用--net=host 选项运行容器时，容器不会在其内部使用独立的网络接口。相反，容器内运行的进程可以直接与宿主机系统的网络接口进行交互。

例如，有一个名为 myrhel-tools 的容器运行了 rhel-tools SPC 镜像，我运行了 docker exec 命令来查看容器内的情况。从那里，我能使用 ip 命令从容器内查看宿主机上的物理网卡（ens3）以及 docker0 网络接口：

```
# docker exec -it myrhel-tools /bin/bash
# ip addr show
...
2: ens3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc...
    link/ether 52:54:00:22:e7:55 brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.224/24 brd 192.168.122.255 scope global...
       valid_lft 2658sec preferred_lft 2658sec
    inet6 fe80::5054:ff:fe22:e755/64 scope link
```

```
       valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500... state UP
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
       valid_lft forever preferred_lft forever
```

仍旧在myrhel-tools容器中，我启动了tcpdump -i命令来监听docker0接口。然后，从宿主机上的另一个容器（未必是SPC），我尝试安装一个RPM软件包。因为这个例子中的SPC可以访问宿主机的docker0接口，所以它能看到fedoraproject.org网站将所请求的软件发送给另一个容器的那些数据包：

```
# tcpdump -i docker0
Utcpdump: verbose output suppressed, use -v or -vv for full...
listening on docker0, link-type EN10MB (Ethernet),...
17:59:30.332411 ARP, Request who-has atomic711.example.com
  tell 172.17.0.10, length 28
17:59:30.332455 ARP, Reply atomic711.example.com is-at 56:84:7a:fe:97:99
(oui Unknown), length 28
17:59:30.332465 IP 172.17.0.10.52036 > 192.168.122.1.domain:
  17163+ A? mirrors.fedoraproject.org. (43)
17:59:30.332747 IP 172.17.0.10.52036 > 192.168.122.1.domain:
  12722+ AAAA? mirrors.fedoraproject.org. (43)
17:59:30.472971 IP 192.168.122.1.domain > 172.17.0.10.52036:
  17163 9/0/0 CNAME wildcard.fedoraproject.org., A 152.19.134.142,
   A 66.135.62.187, A 67.219.144.68, A 209.132.181.15,...
```

13.2.4 访问宿主机的进程间通信

默认情况下，当容器启动时，会为容器创建一个专用的进程间通信（inter-process communication，IPC）命名空间。通过为docker run设置--ipc=host选项，容器会使用宿主机的IPC环境（信号量、消息队列和共享内存）。如果没有将--ipc=设置为host，而是设置为另一个容器的名字，就能够访问那个容器的IPC环境。

要在SPC内部使用宿主机的IPC环境（通过启用--ipc=host），可以使用ipcs命令（获取IPC的信息）、ipcmk命令（创建IPC资源）或者ipcrm命令（删除共享内存ID、消息队列或者信号量集合）。

13.2.5 访问宿主机的文件系统

从容器内开放对宿主机挂载表的访问与开放其他权限略有不同。容器的主要功能之一是其维护了自己的文件系统，因此，容器内运行的命令才具有启动容器化应用所需的一切。

然而，对于SPC，你通常想运行的命令都在容器内（并且宿主机上没有），而后想让这些命令作用于宿主机的文件系统的文件上。例如，你也许想让容器内运行的命令检查宿主机

的配置文件、日志文件或设备。

要实现 SPC 同时查看容器内的文件系统以及宿主机系统上的文件系统，SPC 可以将宿主机文件系统挂载到容器内的/host 目录上。这可以使用-e HOST=/host 选项来完成。

预计随着容器变得更为流行，一些基本的 Linux 系统管理命令会进行修改以照顾到容器。例如，如果在 SPC 内运行 sosreport 命令，该命令会将/host 作为其查询的文件系统的根目录（只从宿主机的文件系统搜集配置数据），而不使用容器自身的根文件系统（/）。

再次重申，谨记-e HOST=/host 选项和刚刚描述的其他权限选项会打破容器与宿主机之间的界限。不应该在生产环境中随意使用这些选项，因为它使容器能够访问宿主机上运行的其他容器以及宿主机自身。

正如我提到的，可以在 docker run 命令中直接使用这些选项。然而，要更高效地使用这些特权功能，请考虑安装 Atomic Host 系统并使用 atomic 命令来安装和运行 SPC，接下来的几节会进行描述。

13.3 准备使用超级特权容器

我使用 RHEL Atomic 系统来尝试本章的 SPC。你可以遵循第 2 章中的指示来获取并运行一个 RHEL Atomic Host 系统。如果没有 RHEL 订阅，可以获取 RHEL Atomic Host 的评估副本或者在 Fedora Atomic Host 上运行这些程序。

本章所述的 SPC 是专为在 Atomic Host 系统上运行而制作的。尽管可以通过附加了选项的 docker run 命令来启动和使用 SPC，但更为简单地运行 SPC 的方法是使用 atomic 命令。

13.4 使用 atomic 命令

docker 命令对于手工处理单个容器而言是非常出色的工具。为了自动运行容器和连接到宿主机系统，已经开发出许多其他工具来处理 Docker 格式的容器。atomic 命令就是这样一个工具，Fedora、RHEL 和 CentOS 的 Atomic Host 中都附带有 atomic 命令。

创建 atomic 命令是为了在 Atomic Host 系统上安装和运行容器化应用。要使用 atomic，需要使用下面的参数运行该命令。

- **install**：将所选的容器镜像拉取到本地系统。当运行这个命令时，还能为保存的镜像指定一个不同的镜像和容器名。该选项与 docker pull 命令本质上是一样的。
- **info**：获取所选容器镜像的信息，帮助识别容器包含什么、从哪里来以及怎样运行它。如果容器不在本地系统中，atomic 会尝试从任何启用的 repository 中拉取镜像。
- **run**：通过从容器镜像中的 LABEL RUN 行识别出的命令行来运行所选的容器。如果将你自己的命令添加到 atomic run 命令行，LABEL RUN 行的设置会被忽略。
- **uninstall**：从本地系统卸载容器镜像。它将依据容器中 LABEL UNINSTALL 行的

内容进行卸载。

- **update**：从 repository 中拉取所选镜像的最新版本来替换当前版本。

atomic 命令对于运行 SPC 特别有用。这是因为 atomic 能够读取从容器自身内部识别的 Docker 元数据，以此打开所需的命名空间来运行一个具有作用于宿主机系统特权的容器。

在启动 RHEL Atomic Host 系统并登录后，这里有一些 atomic 命令的例子，你可以运行这些命令来处理 SPC。对于这些示例，我使用了 rhel-tools 容器，rhel-tools 容器将在本章后面详细描述。



注意

rhel-tools 容器镜像大概有 1 GB 大小，因此，要确保 Atomic Host 上有足够的空间来存放该容器镜像以及系统上的其他容器软件。

13.4.1 用 atomic 安装 SPC 镜像

你可以像下面这样（从 RHEL 或 Fedora 系统）使用 atomic install 命令将 rhel-tools 的 SPC 拉取到自己的 RHEL Atomic 系统中：

```
# atomic install rhel7/rhel-tools
Trying to pull repository registry.access.redhat.com/rhel7/rhel-tools
9a8ad4567c27: Pulling image (latest) from registry.access.redhat.com/
rhel7/rhel-9a8ad4567c27: Pulling image (latest) from registry.access.
redhat.com/rhel7/rhel-9a8ad4567c27: Download complete
Status: Downloaded newer image for registry.access.redhat.com/rhel7/
rhel-tools:latest
```

要核实该镜像现已在你的本地系统中可以使用，可以输入下面的命令：

```
# docker images | grep rhel-tools
REPOSITORY                                    TAG        IMAGE ID
   CREATED          VIRTUAL SIZE
registry.access.redhat.com/rhel7/rhel-tools 7.1-9      9a8ad4567c27
   9 weeks ago      994.8 MB
registry.access.redhat.com/rhel7/rhel-tools latest     9a8ad4567c27
   9 weeks ago      994.8 MB
```

该镜像（9a8ad4567c27）被打了两个标签，一个是“latest”标签，另一个标签表明了该容器镜像是在哪个 RHEL 发行版中构建的。

13.4.2 用 atomic 获取 SPC 镜像的信息

要查看这个容器或者其他容器构建来用于 atomic 命令的信息，可以运行 atomic info

命令。只有 SPC 以及其他为用于 atomic 命令而做好准备的镜像才会显示该信息。本示例展示了 rhel-tools 容器的信息：

```
# atomic info rhel7/rhel-tools
RUN          : docker run -it --name NAME --privileged --ipc=host
                 --net=host --pid=host -e HOST=/host -e NAME=NAME
                      -e IMAGE=IMAGE -v /run:/run -v /var/log:/var/log
                  -v /etc/localtime:/etc/localtime -v /:/host IMAGE
Name         : rhel7/rhel-tools
Build_Host   : rcm-img01.build.eng.bos.redhat.com
Version      : 7.1
Architecture : x86_64
Release      : 15
Vendor       : Red Hat, Inc.
BZComponent  : rhel-tools-docker
Authoritative_Registry: registry.access.redhat.com
```

从 atomic info 的输出中可以看到，你能像对待 RPM 或 Deb 软件包那样对待容器镜像。这些信息让你了解到谁创建了这个镜像、构建的主机、版本信息以及架构和发布信息。RUN 行显示了实际的 docker 命令，如果你没有给 atomic run 命令提供其他选项来启动容器镜像，就会运行该命令。我会在稍后探讨这些选项。

13.4.3 用 atomic 运行 SPC 镜像

atomic run 命令启动所选容器并使其在后台继续运行。它是通过读取容器内 LABEL RUN 的值并执行该值所包含的完整 docker 命令行来完成的。如果镜像没有包含 LABEL RUN 的值，atomic run 会使用默认行为，我稍后会描述。

通常，会给 atomic run 命令传递两个参数：为正在运行的容器选择的名字（--name=）以及镜像的名字（这个例子中是 rhel7/rhel-tools）。下面是使用 atomic run 启动 rhel-tools SPC 的例子：

```
# atomic run --name myrhel-tools rhel7/rhel-tools
docker run -it --name myrhel-tools --privileged --ipc=host --net=host
--pid=host -e HOST=/host -e NAME=myrhel-tools -e IMAGE=rhel7/rhel-tools
-v /run:/run -v /var/log:/var/log -v /etc/localtime:/etc/localtime -v
/:/host rhel7/rhel-tools

#
```

你能看到 atomic 启动的 docker 命令行。注意，你传递给 atomic 的容器名和镜像名包含在 docker 命令中。默认操作是在容器内打开 bash shell。

随着在容器内打开 bash shell，此时输入的任何命令都会在容器内运行。但是，因为此时这个容器打开了许多宿主机特性，要判断你在容器内可能比较困难。

尝试在容器内输入下面的命令：

```
# ls /
bin   dev home lib   media opt    root sbin sys usr
boot  etc host lib64 mnt   proc   run  srv  tmp var
# ls /host
bin   dev home lib64 mnt   ostree root sbin sys      tmp  var
boot  etc lib  media opt   proc   run  srv  sysroot  usr
# exit
```

/host 目录应该会出现在 SPC 中。列出的/host 目录的内容（ls /host），应该看到宿主机文件系统的根，其应该包含/ostree 目录（对于 RHEL Atomic Host 系统）。你可以继续在容器内运行想运行的命令。完成之后，输入 exit 退出容器并返回到宿主机系统的 shell。

除了打开 shell，你还可以使用自己选择的任何命令来运行 SPC。例如，man 命令并没有包含在 RHEL Atomic Host 中，所以你无法在宿主机上阅读 docker 命令的联机帮助页。然而，rhel-tools 容器包含有 man 命令和 docker 的联机帮助页。因此，要查看 docker 的联机帮助页，可以输入下面的命令：

```
# atomic run rhel7/rhel-tools man docker
DOCKER(1)          APRIL 2014                    DOCKER(1)
NAME
  docker - Docker image and container command line interface
SYNOPSIS
       docker [OPTIONS] COMMAND [arg...]
DESCRIPTION
  docker  has two distinct functions. It is used for starting the
Docker daemon and to...
```

13.4.4 用 atomic 停止和重启 SPC

用完一个打开 shell 的容器后，只要输入 exit 就可以退出。对于通过 atomic run 命令启动并在后台运行的容器，可以使用 docker stop 命令来停止。例如：

```
# docker stop myrhel-tools
```

刚刚展示的 docker 命令停止了正在运行的容器，但并没有自动删除它（除非镜像的 RUN LEVEL 值为 docker 包含了--rm 选项）。如果想使用这个容器镜像，可以使用 docker start 命令重启它：

```
# docker start -i myrhel-tools
```

该容器（不是原始镜像）会再次启动并准备好使用。

13.4.5 更新 SPC 镜像

每过一段时间，就会有新版本的 SPC 镜像可用。要用 atomic 命令更新 rhel-tools 镜像，输入下面的命令：

```
# atomic update rhel7/rhel-tools
Trying to pull repository registry.access.redhat.com/rhel7/rhel-tools ...
9a8ad4567c27: Download complete
Status: Image is up to date for registry.access.redhat.com/rhel7/
➥rhel-tools:latest
```

13.4.6 卸载 SPC 镜像

如果使用完 SPC 镜像，可以用 atomic uninstall 命令删掉它。例如，要将 rhel-tools 镜像从系统中删除，需要输入下面的命令：

```
# atomic uninstall rhel7/rhel-tools
```

运行 atomic uninstall 不仅可以从本地系统删除镜像，还可以做更多事情。在某些情况下，atomic uninstall 运行脚本来修改或删除宿主机系统上的文件，想必是让一切回到 SPC 安装之前。可以在 SPC 上使用 atomic info 来查看是否存在卸载动作。

现在，已经了解了 docker 命令可以使用的特权选项，也知道了这些选项如何与 atomic 命令一起使用。接下来，我要讲述几个可用于 RHEL Atomic 的 SPC，可以用它们管理 RHEL 或 Fedora 的 Atomic Host，或者一同使用。

13.5 尝试一些 SPC

在 Red Hat 的容器 registry（registry.access.redhat.com）中，可以找到若干专为增加 RHEL Atomic 系统的功能而设计的 SPC。如果你计划将容器部署到大型企业环境中，这些 SPC 能够帮你将一系列故障诊断和监控工具拖入已部署的容器中。

期待在不久的将来会有更多 SPC 使你能够为运行 Docker 格式容器的宿主机增加功能。此刻，如果你有 RHEL Atomic 或 RHEL 7 宿主机（默认情况下，它们配置了从 Red Hat 的容器 registry 中拉取），可以尝试本节描述的这些 SPC。Fedora Atomic Hosts 上也有这些容器，而且无需指向额外的 registry。

这里描述的 SPC 包括下面这些。

- **RHEL 工具容器（rhel-tools）**：包含许多用于宿主机系统的故障诊断与调试的工具。
- **系统日志容器（rsyslog）**：包括 rsyslog 守护进程以及配置文件，用于收集消息以及将消息重定向到所选的文件或日志宿主机上。
- **系统监控容器（sadc）**：包含系统活动数据收集器（sadc），监控并收集系统活动数据。它还包含来自 sysstat 软件包的其他软件，以及用于展示所收集数据的命令，如 sar 命令。

13.5.1 运行 RHEL 工具 SPC

rhel-tools 容器是 SPC 的“瑞士军刀”。它包含了近 400 个 RHEL Atomic Host 系统没

有的命令。其目的是为了宿主机的故障诊断和其他检查包含尽可能多的有用工具。如果工具不存在，可以使用 yum 命令向容器添加更多软件包。

为了获取正确的选项以便 Docker 将 rhel-tools 作为 SPC 运行，按照之前 13.4 节所介绍的，应该使用 atomic 命令来运行它。使用该容器无需专门配置。

下面是一些命令和特性的例子，rhel-tools 通过这些命令和特性增加了你使用 RHEL Atomic 系统的能力。

- **文档**：为了节省空间，RHEL Atomic Host 移除了所有文档。rhel-tools 容器通过包含从/usr/share/doc 目录剥离的文档、联机帮助页和信息文本为宿主机提供可用的文档。特别是，你可以找到与容器组件相关的联机帮助页，如 docker 和 Kubernetes（kubectl 命令）。
- **网络诊断工具**：rhel-tools 容器添加了诊断网络问题的常用工具，包括 tcpdump、tcpslice、netsniff-ng 和 nfsiostat。
- **应用程序调试工具**：rhel-tools 容器增加了用于应用程序调试的命令，如 strace、stap（Systemtap 的运行时工具）以及 ltrace 命令。
- **块设备故障诊断工具**：rhel-tools 中用于检查、监控和修正块设备问题的工具有 blkiomon、blkparse、blktrace、blkrawverify 和 verify_blkparse。

rhel-tools 容器中还有数百个其他命令。rhel-tools 容器中的有些命令已经修改，以使其知晓自己正在容器中运行。例如，当 sosreport 命令从容器内运行时，它知道使用 /host 作为文件系统的根，而不是用/，所以它会收集宿主机系统的数据而不会收集容器文件系统的数据。

13.5.2 运行日志记录（rsyslog）SPC

通过将 rsyslog 容器添加到 Atomic Host 系统中，可以提供一个集中的位置收集来自宿主机系统的日志消息，或者是来自宿主机系统上的容器的日志消息。要安装 rsyslog 容器，输入下面的命令：

```
# atomic install rhel7/rsyslog
Trying to pull repository registry.access.redhat.com/rhel7/rsyslog ...
b5168acccb4c: Download complete
Status: Downloaded newer image for registry.access.redhat.com/rhel7/
➥rsyslog:latest
docker run --rm --privileged -v /:/host -e HOST=/host -e IMAGE=rhel7/
➥rsyslog -e NAME=rsyslog rhel7/rsyslog /bin/install.sh
Creating directory at /host//etc/pki/rsyslog
Installing file at /host//etc/rsyslog.conf
Installing file at /host//etc/sysconfig/rsyslog
```

对于 rsyslog 容器镜像，atomic install 命令不只是下载该镜像，还运行一个脚本（/bin/install.sh）——这个脚本会设置容器运行所需的文件。尤其是，它在容器和宿主

机之间共享配置文件（如/etc/rsyslog.conf 和/etc/sysconfig/rsyslog），这样就能在宿主机上处理这些文件，而从容器内运行 rsyslogd 守护进程。

在容器下载并安装之后，可以按如下方式运行容器镜像：

```
# atomic run rhel7/rsyslog
docker run -d --privileged --name rsyslog --net=host -v /etc/pki/
rsyslog:/etc/pki/rsyslog -v /etc/rsyslog.conf:/etc/rsyslog.conf -v
/etc/rsyslog.d:/etc/rsyslog.d -v /var/log:/var/log -v /var/lib/rsyslog:
/var/lib/rsyslog -v /run/log:/run/log -v /etc/machine-id:/etc/machine-id
-v /etc/localtime:/etc/localtime -e IMAGE=rhel7/rsyslog -e NAME=rsyslog
--restart=always rhel7/rsyslog /bin/rsyslog.sh
36739324e1e1dd918aa2c93765a9de9201c43d816d71e2e5c16c80c5e38ea9c5
```

如果查看运行该容器的 docker 命令行，可以看到最后的选项运行了/bin/rsyslog.sh 脚本，该脚本使用适当的选项运行 rsyslogd 守护进程。rsyslogd 服务作为守护进程（-d）在后台运行，并将宿主机的几个文件和目录挂载到容器内部（/etc/rsyslog.conf、/etc/rsyslog.d/、/var/log/、/var/lib/rsyslog/以及其他文件和目录）。

--privileged 选项将宿主机的权限开放给容器，而且容器能够看到宿主机的网络接口（--net=host）。restart=always 选项确保系统重启时容器会重新启动。

想要查看该 SPC 的信息，可以按如下方式使用 atomic info 命令：

```
# atomic info rhel7/rsyslog
RUN : docker run -d --privileged --name NAME --net=host
      -v /etc/pki/rsyslog:/etc/pki/rsyslog
      -v /etc/rsyslog.conf:/etc/rsyslog.conf
      -v /etc/rsyslog.d:/etc/rsyslog.d -v /var/log:/var/log
      -v /var/lib/rsyslog:/var/lib/rsyslog
      -v /run/log:/run/log -v /etc/machine-id:/etc/machine-id
      -v /etc/localtime:/etc/localtime -e IMAGE=IMAGE
      -e NAME=NAME --restart=always IMAGE /bin/rsyslog.sh
Name         : rsyslog-docker
Build_Host   : rcm-img04.build.eng.bos.redhat.com
Version      : 7.1
Architecture: x86_64
INSTALL      : docker run --rm --privileged -v /:/host -e HOST=/host
               -e IMAGE=IMAGE -e NAME=NAME IMAGE /bin/install.sh
Release      : 3
Vendor       : Red Hat, Inc.
```

注意，当你安装镜像和运行镜像（RUN 所在行）时会使用不一样的 docker 命令行。atomic info 的其他信息展示了关于容器的类似 RPM 的信息，如容器是在哪个系统上构建的（Build_Host）、RHEL 的版本（7.1），以及是为哪种体系结构构建的（x86_64）。

要检查 rsyslogd 服务是否正在运行，在宿主机上运行 logger 命令并发送一条消息，可以使用 journalctl 命令检查该消息：

```
# logger "Testing that rsyslog is boffo"
```

```
# journalctl | grep boffo
Apr 27 22:43:32 node1 root: Testing that rsyslog is boffo
```

要查看或更改日志消息的定向方式，可以编辑宿主机系统上的/etc/rsyslog.conf 文件。要密切关注宿主机上/var/log 目录的磁盘空间使用情况。

13.5.3 运行系统监控（sadc）SPC

sysstat 软件包(包含了 sar 和 iostat 命令)和用来监控系统性能的 sadc 工具自 UNIX 时代就已经存在了。该软件包包含了一个标准工具集——这些工具可以持续监控各种组件(网络接口、块设备、CPU 使用等）的系统性能。将 sadc 容器添加到 RHEL Atomic Host 能够收集和展示宿主机系统的性能数据。

按如下方式从RHEL Atomic Host系统运行 atomic install 命令，可以从RHEL registry 拉取 sadc 容器并将其安装到本地系统：

```
# atomic install rhel7/sadc
Trying to pull repository registry.access.redhat.com/rhel7/sadc ...
1a97a9cc4d1b: Download complete
Status: Downloaded newer image for registry.access.redhat.com/rhel7
➥ /sadc:latest

docker run --rm --privileged --name sadc -v /:/host -e HOST=/host
➥ -e IMAGE=rhel7/sadc -e NAME=name rhel7/sadc /usr/local/bin/
➥sysstat-install.sh
Installing file at /host//etc/cron.d/sysstat
Installing file at /host//etc/sysconfig/sysstat
Installing file at /host//etc/sysconfig/sysstat.ioconf
Installing file at /host//usr/local/bin/sysstat.sh
```

除了拉取 sadc 容器，还可以看到有几个配置文件和脚本被安装到宿主机系统上，以便在容器运行的时候给予支持。使用 atomic info 命令，可以看到容器安装（INSTALL）和运行（RUN）时使用的完整的 docker run 命令行。

```
# atomic info rhel7/sadc
RUN : docker run -d --privileged --name NAME
      -v /etc/sysconfig/sysstat:/etc/sysconfig/sysstat
      -v /etc/sysconfig/sysstat.ioconf:/etc/sysconfig/sysstat.ioconf
      -v /var/log/sa:/var/log/sa -v /:/host -e HOST=/host
      -e IMAGE=IMAGE -e NAME=NAME --net=host --restart=always
      IMAGE /usr/local/bin/sysstat.sh
Name         : sadc-docker
License      : GPLv3
Build_Host   : rcm-img05.build.eng.bos.redhat.com
Version      : 7.1
Architecture : x86_64
INSTALL      : docker run --rm --privileged --name NAME -v /:/host
               -e HOST=/host -e IMAGE=IMAGE -e NAME=name IMAGE
```

```
                /usr/local/bin/sysstat-install.sh
Release       : 3
Vendor        : Red Hat, Inc.
UNINSTALL     : docker run --rm --privileged -v /:/host -e HOST=/host
                -v /var/log:/var/log -e IMAGE=IMAGE -e NAME=NAME
                IMAGE /usr/local/bin/sysstat-uninstall.sh ;
                docker rm -f sadc
```

如果已经准备好使用 sadc 容器，像下面这样使用 atomic run 命令启动它：

```
# atomic run rhel7/sadc
docker run -d --privileged --name sadc -v /etc/sysconfig/sysstat:/etc/
sysconfig/sysstat -v /etc/sysconfig/sysstat.ioconf:/etc/sysconfig/
sysstat.ioconf -v /var/log/sa:/var/log/sa -v /:/host -e HOST=/host
-e IMAGE=rhel7/sadc -e NAME=sadc --net=host --restart=always rhel7/
sadc /usr/local/bin/sysstat.sh
c02b44ce7e4494571a7da13113a47873fd80d8d38201cbdb1e6c898c7df5caca
```

一旦 sadc 容器开始运行，它就已经准备好开始收集系统数据了。数据收集是通过宿主机上的 crond 服务启动的，使用了/etc/cron.d/sysstat 文件。看一下这个文件的内容：

```
# Run system activity accounting tool every 10 minutes
# Customized for sadc container
*/10 * * * * root docker exec -d sadc /usr/lib64/sa/sa1 1 1
# 0 * * * * root /usr/lib64/sa/sa1 600 6 &
# Generate a daily summary of process accounting at 23:53
# Customized for sadc container
53 23 * * * root docker exec -d sadc /usr/lib64/sa/sa2 -A
```

基于这个文件的内容，可以了解到系统活动数据每 10 分钟收集一次，每天晚上 11:53 会生成每天的报告。如果你熟悉 syssta 软件包，会注意到这个文件已经为通过 docker exec 命令运行 sadc 工具而做了修改。

要改变数据的收集方式，可以修改/etc/cron.d/sysstat 文件。参考 sadc 和 crontab（第 5 节）的联机帮助页来获取该文件格式的信息。然而，由于你所在的 RHEL Atomic 系统没有联机帮助页，你可以使用 rhel-tools 容器来读联机帮助页：

```
# docker run --rm rhel7/rhel-tools man 5 crontab
```

要查看系统活动数据，可以在容器内使用 sar 命令。例如：

```
# docker exec -it sadc sar
Linux 3.10.0-229.1.2.el7.x86_64 (node1) 04/28/15 _x86_64_ (1 CPU)
08:57:15      LINUX RESTART
09:00:01    CPU   %user   %nice   %system   %iowait   %steal   %idle
09:10:02    all    1.53    0.00      1.07      0.62     0.01   96.78
09:20:02    all    0.78    0.00      0.87      0.67     0.01   97.68
09:30:01    all    0.81    0.00      0.93      0.72     0.01   97.53
09:40:01    all    1.29    0.00      1.43      1.34     0.01   95.93
Average:    all    1.10    0.00      1.08      0.84     0.01   96.98
```

随着 Atomic 项目开发继续进行，期望看到更多像 rhel-tools、rsyslog 和 sadc 这样的 SPC 涌现出来，用来监控、故障诊断和管理 Atomic Host 系统。

13.6 小结

由于容器运行时操作系统（如 RHEL Atomic）的设计方式，它们没有包含管理和维护这些系统可能需要的所有工具。对于 Atomic 项目，这个问题是通过使用超级特权容器（SPC）来解决的。

SPC 使你能够很容易地向一个精简的、面向容器的操作系统添加软件，或者从中移除软件。可以通过为 docker run 命令添加选项来创建 SPC。然而，已经为 RHEL Atomic Host 创建了几个可以使用专门的 atomic 命令进行管理的 SPC。

atomic 命令设置了 docker run 的选项，而这些选项明确定义了要开放宿主机的哪些权限和命名空间。它还在容器安装时完成这样的工作，如复制文件到宿主机，以及将目录从宿主机挂载到容器中。本章对 3 个 SPC 示例进行了详细说明。通过 rhel-tools 容器可以向 Atomic Host 添加许多故障诊断和维护命令。rsyslog 容器为宿主机添加了日志设施。sadc 容器添加了 sysstat 软件包来从宿主机系统收集系统活动数据。

你可以在云上部署的 Atomic Host 系统上运行来自 SPC 的工具，这可以修复和监控云上运行的基于容器的操作系统。接下来的一章会帮你了解各种可以用于构建自己的 SPC 的不同的工具，以及其他用来在云环境中部署容器的工具。

第 14 章

用 Cockpit 管理云中的容器

本章主要内容

- 了解 Cockpit 如何管理面向容器的操作系统
- 安装与启动 Cockpit
- 添加服务器让 Cockpit 进行管理
- 在 Cockpit 中处理容器
- 在 Cockpit 中配置网络与存储
- 使用 Cockpit 工具添加用户权限以及打开终端窗口

一旦已经习惯了启动 Atomic、Fedora、RHEL 或其他操作系统并在其上运行容器，你会发现自己在云环境中或在虚拟化主机上运行了多个系统。单独管理这些系统会成为一种负担。与其使用 ssh 这样的工具远程登录每个系统，不如使用 Cockpit 这样的工具通过一个界面管理多个系统。

Cockpit 提供了一个基于 Web 的用户界面来管理 Fedora、RHEL、Atomic 和类似的 Linux 系统。因为它包含的工具既可以监控单个容器，也可以监控由 Kubernetes 编排而成的成组容器，所以 Cockpit 特别擅长管理那些专为容器化应用部署的系统。此外，Cockpit 还包含一些用于通用系统管理的优秀特性。

如果到目前为止，直接使用 Linux 命令行进行工作对你而言还是一项挑战的话，我告诉你一个好消息。Cockpit 设计了一个简单易用的点选式界面来探索和处理像 Docker 容器和 Kubernetes 集群这样的复杂技术。

本章中展示了如何从 Fedora 系统运行 Cockpit。接着，仅通过一个 Cockpit 界面，就可以添加尽可能多的面向容器的系统（运行了 Atomic、RHEL 或 Fedora），对这些系统上的容器进行监控、启动、停止以及其他操作。



注意

Cockpit 仍处于发展的早期阶段。本书写作时，其版本尚未达到 1.0，这表明要使其达到稳定仍有许多工作要做。我在这里展示它，是为了让读者能够对专为管理单个容器或集群容器而开发的这类简单前端有个体会。随着底层容器特性持续稳固，可以预期 Cockpit 和类似工具会使这些特性的使用变得更容易。

14.1 了解 Cockpit

Cockpit 作为一个服务运行，监听 TCP 9090 端口的请求。要使用该服务，只要打开 Web 浏览器，访问运行 Cockpit 的系统的 9090 端口并以管理员用户登录（最开始通常是 root 用户）。

当初次以 root 登录时，可以将服务管理员（Server Administrator）权限分配给拥有 Cockpit 所在系统账号的其他用户。或者，可以直接以 root 用户身份开始使用 Cockpit。

一旦登录到 Cockpit，就可以开始监控本地系统的各种特性，也可以添加要监控的其他系统。要添加其他系统，只要提供主机名（或 IP 地址）、管理用户名（如 root）以及该用户的口令。

如果你已经把 Atomic、Fedora、RHEL 或其他类似系统推送到云上或者推送到运行虚拟机的主机上（如 Linux KVM 环境），那么只要用一个 Cockpit 仪表盘就可以监管这些系统以及运行在这些系统上的容器。通过 Cockpit，你可以选择想要查看的系统，而后通过左侧边栏查看下面这些类型的信息。

- System（系统）：这个选项卡监控所选系统的资源消耗。通用信息有硬件类型、BIOS、主机名和其他基本信息。你可以监视 CPU、内存、网络流量和磁盘 I/O 资源的渐进使用情况。图 14-1 中的 System 选项卡展示了直接安装在戴尔硬件上的 Red Hat Enterprise Linux 系统。图 14-2 是一个安装在 KVM 虚拟机上的 Fedora 的例子。

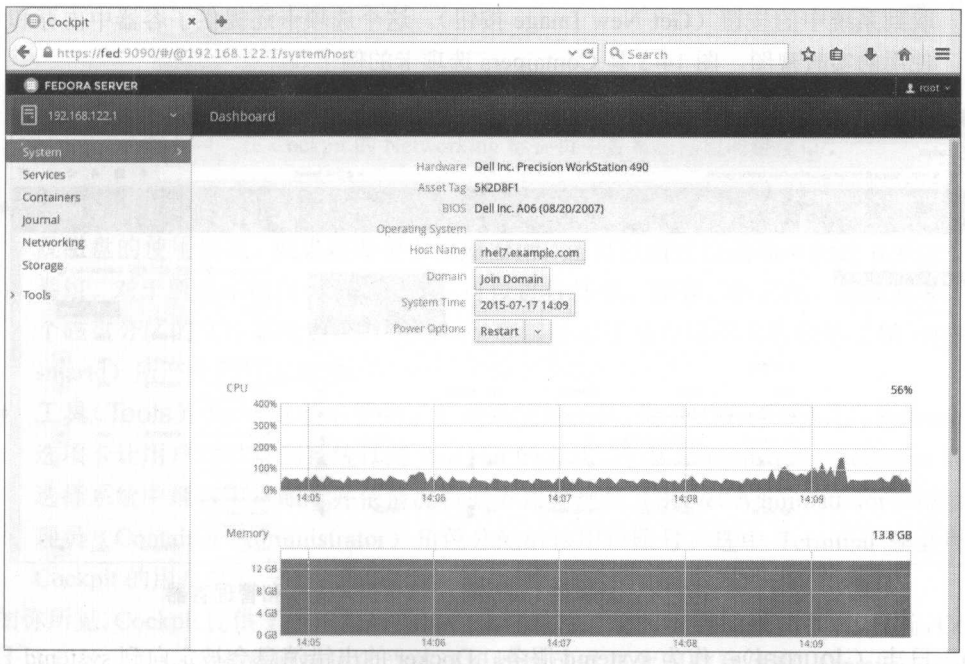


图 14-1 Cockpit 显示了戴尔硬件上的 RHEL 7 宿主机的信息

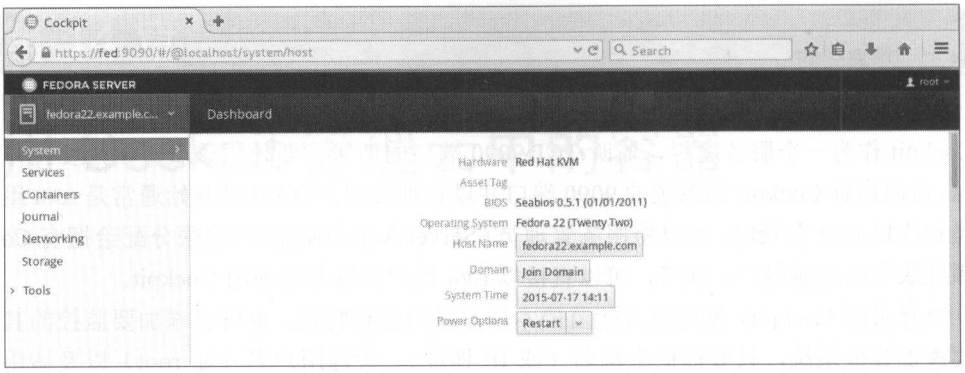


图 14-2 Cockpit 显示了 KVM 宿主机上的 Fedora 虚拟机的信息

- **Services（服务）**：通过 Services 选项卡，可以看到所选宿主机上的全部 systemd 资产。从所选的系统中，可以看到有哪些启用和禁用的系统服务正在运行。还可以选择查看系统上配置的 Targets（成组服务）、Sockets（服务通信的端点）、Timers（用来设置特定时间的事件的单元文件）和 Paths（与文件/目录被访问时要执行的操作相关联的单元文件）。
- **容器（Containers）**：通过 Container 选项卡管理所选系统上的容器。可以看到系统中每一个正在运行的容器或停止的容器。一个滑动条显示了已消耗的存储空间和可用的存储空间。Images 框中显示了系统中的所有容器镜像，以及一个可以将新镜像拉取到系统中的按钮（Get New Image 按钮）。这个选项卡还提供了容器中内存和 CPU 使用的实时视图。图 14-3 是 Containers 选项卡的例子。

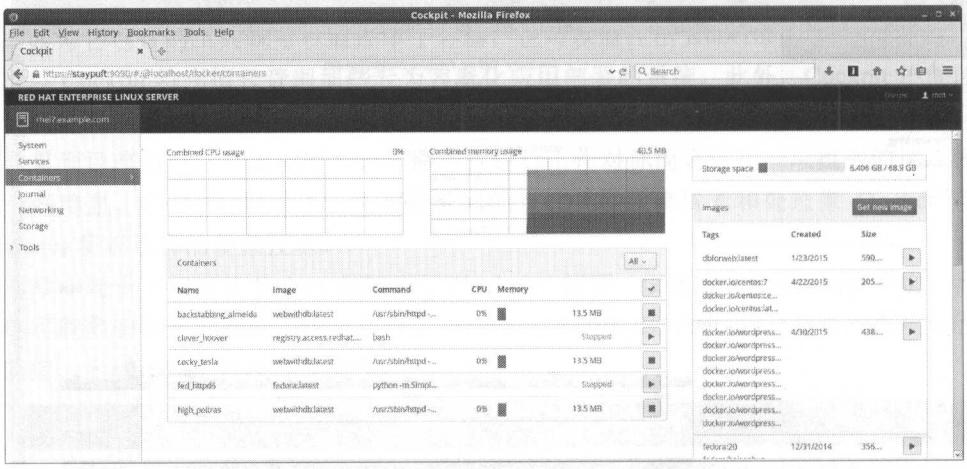


图 14-3 通过 Cockpit 的 Containers 标签页来查看和管理容器

- **日志（Journal）**：作为 systemd 服务，Docker 的出错消息会被定向到 systemd 日志。可以通过 Journal 选项卡查看 Docker 的日志消息。可以通过日期和严重程度（Errors、

Warnings、Notices 或 All）来选择消息。选中消息可以查看该消息的详情。

- **网络（Networking）**：查看与所选宿主机相关联的网络接口的活动。图 14-4 展示了一个拥有多个网络接口的物理计算机上的网络接口情况。docker0 接口提供了对系统中容器的网络访问。系统有两个物理网卡（enp11s0 和 enp12s2），其中一个已启用并提供对外部网络的访问。因为系统正在运行虚拟机，所以 virbr0 网络接口被定义为默认的虚拟机网络。该选项卡的顶部展示了网络发送和接收的流量，底部是网络日志项。

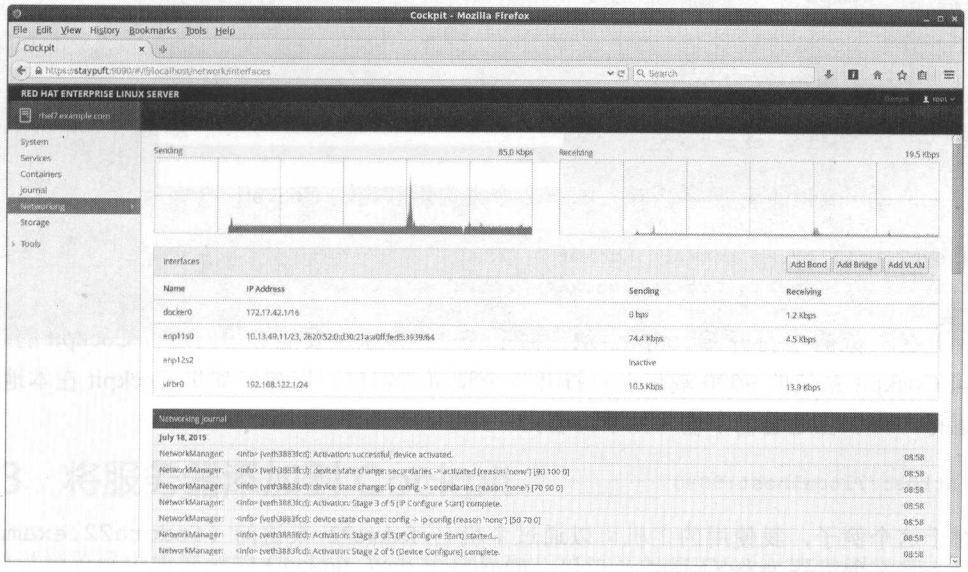


图 14-4 在 Cockpit 的 Networking 标签页中查看和修改网络接口

- **存储（Storage）**：Cockpit 的 Storage 选项卡可以监控所选宿主机的可用磁盘空间以及磁盘的使用情况。如果需要更多存储空间，还可以通过该选项卡创建 RAID 设备或卷组。对于所显示的每个可用磁盘，可以进行挂载、卸载、格式化、删除或者修改每个磁盘分区的文件系统选项。该选项卡底部显示了与存储相关的服务（如 udiskd 和 smartd）所产生的日志消息。
- **工具（Tools）**：Tools 标签下罗列了管理系统的工具。对 RHEL 系统而言，Subscriptions 选项卡让用户可以看到系统订阅 Red Hat 的方式。选中 Administrator Accounts 就可以选择系统中现有用户账号并依情况将服务器管理员（Server Administrator）或容器管理员（Container Administrator）角色分配给该用户账号。选中 Terminal 就会用登录 Cockpit 的用户打开一个所选系统的 shell。

如你所见，Cockpit 提供了一组很好的特性来管理用于运行容器的操作系统。目前，Cockpit 还处于活跃的开发中。本章所描述的当前这组特性可以帮你跟踪和处理 Atomic、RHEL、Fedora 和 CentOS 系统上的单个容器。在第 16 章，我将描述 Cockpit 正在开发的一些新特性，

这些特性可以管理 Kubernetes 编排的容器集合。

14.2 着手使用 Cockpit

目前，在 Fedora 和 Red Hat Enterprise Linux 系统上都有 cockpit 软件包可用。本节描述如何在 Fedora 上安装和运行 Cockpit，相同的过程在 Red Hat Enterprise Linux 上也适用（前提是你有 RHEL 系统并且该系统有适当的订阅）。

（1）**安装 Fedora**。安装最新版的 Fedora（这个例子中使用的是 Fedora 22）。

（2）**安装 cockpit 软件包**。使用 root 用户（或使用 sudo），按如下方式安装 cockpit 软件包（以及一些依赖的软件包）：

```
# yum install cockpit -y
```

（3）**启动 cockpit 套接字**。输入如下命令来启用和启动 Cockpit：

```
# systemctl start cockpit.socket
# systemctl enable cockpit.socket
```

（4）**在浏览器中打开 Cockpit**。从任何系统——该系统要能够访问运行 Cockpit 的系统，并访问 Cockpit 系统的 9090 端口——打开一个浏览器窗口。例如，如果 Cockpit 在本地宿主机上运行，可以在地址栏中输入如下内容：

```
https://localhost:9090
```

对于这个例子，我使用的主机可以通过主机名 fedora22 来访问（`fedora22.example.com:9090`）。图 14-5 是该主机的 Cockpit 登录界面。



注意

如果运行 Cockpit 的系统上有已启用的防火墙，并且该防火墙阻止了对该系统上的端口的访问，就需要打开 TCP 9090 端口。如果主机的防火墙是基于 iptables 的，要暂时这样做的话，可以运行下面的命令来开放 TCP 9090 端口。如果运行这个命令后该端口可以访问，就可以查看系统文档了解如何让防火墙永久性地开放这个端口：

```
# iptables -A INPUT -p tcp -m tcp --dport 9090 -j ACCEPT
```

（5）**登录到 Cockpit**。使用 Cockpit 所在系统的 root 账号或是拥有 Cockpit 服务器管理员权限的其他用户登录到 Cockpit。如果运行 Cockpit 的系统还运行了 Docker，可以立即开始管理该系统上的容器。如果没有，接下来要做的就是启动 Docker 并向 Cockpit 添加一个或多个要管理的 Docker 系统。

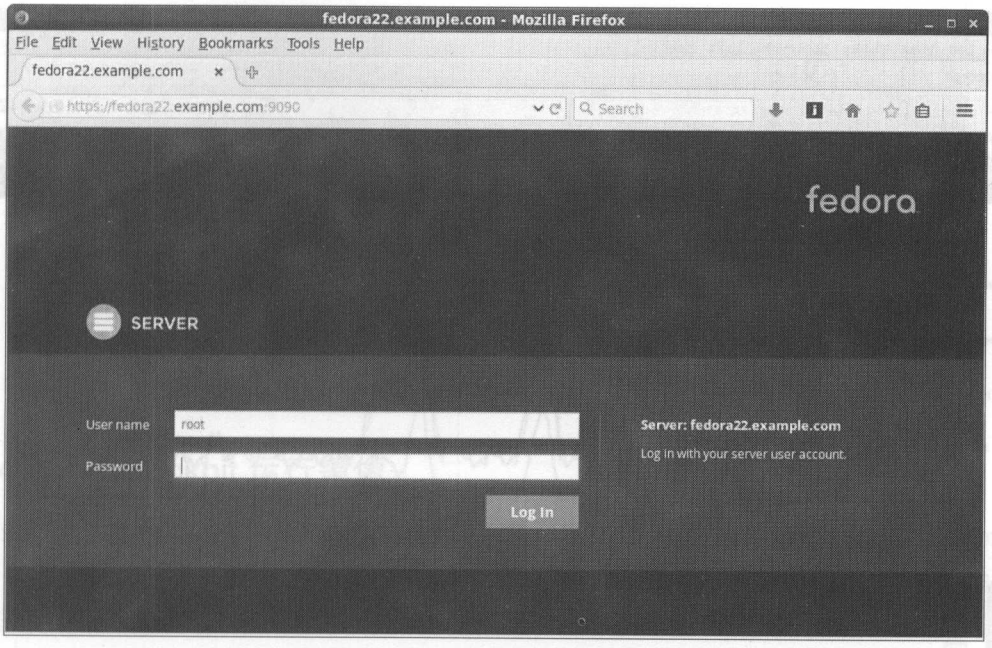


图 14-5 从 Cockpit 系统的 9090 端口访问登录界面

14.3 将服务器添加到 Cockpit

通过浏览器中所展示的 Cockpit Web 用户界面，可以开始将 Docker 容器添加到 Cockpit 中。遵循下面这些步骤将容器服务器系统添加到 Cockpit 中。

（1）**打开 Dashboard**。选择顶部的 Dashboard 标签页。应该会看到有关系统资源使用情况的滚动信息以及位于屏幕底部的服务器列表（目前可能只有本地系统）。

（2）**添加服务器**。选择位于 Servers 选项卡的添加服务器按钮（+）并填写服务器的主机名或 IP 地址。如果新服务器使用相同的认证信息(用户名和密码)，保持 Log in with My Current Credentials 复选框被选中。否则，取消勾选。然后点击 Next。

（3）**添加认证信息**。在弹出的 Add Host 窗口中（如果出现的话），输入访问新服务器所需的用户名和密码。这应该是一个特权用户（如 root）。然后点击 Next。

（4）**添加宿主机**。如果可以找到服务器并使用认证信息访问该服务器，宿主机的地址和特征信息就会被显示。如果看上去没错，选择 Add Host 按钮。图 14-6 显示了额外添加两个服务器到 fedora22 宿主机后的 Cockpit Dashboard，一个通过 IP 地址（192.168.122.1）添加而另一个通过主机名（rhel7.example.com）添加。

选择 Dashboard 页面上所列的服务器，就可以开始通过 Cockpit 管理这些服务器了，连同管理这些系统上运行的容器。



图 14-6 将想监控的服务器添加到 Cockpit Dashboard 中

14.4 从 Cockpit 管理容器

从 Cockpit 的 Containers 选项卡，可以开始使用所选系统上的容器。系统中已存在的镜像或正在运行的容器会出现在 Containers 区域。如果想开始添加自己的镜像和正在运行的容器，可以直接从 Cockpit 完成。

14.4.1 向 Cockpit 添加容器镜像

从 Containers 选项卡，选中 Get New Image 按钮。输入一个搜索关键字来搜索该系统启用的 registry（至少是 Docker Hub Registry，以及其他可能的 registry）。图 14-7 展示的示例是对名为 cricket/hangman 的镜像进行的搜索。

在搜索结果中选中想要的镜像并点击 Download 将该镜像下载到所选的服务器中。下载的镜像会出现在 Containers 选项卡的 Images 列表中。

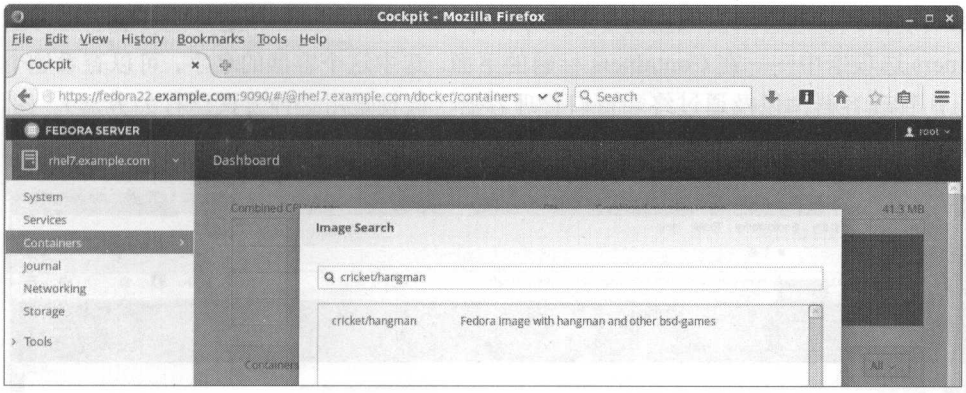


图 14-7 从 Cockpit 的 Containers 标签页搜索要下载的镜像

14.4.2 从 Cockpit 运行镜像

将镜像下载到 Cockpit 管理的服务器中后，还可以从 Cockpit 运行该镜像。从 Containers 选项卡顶部的 Images 列表中找到想要的镜像，点击与该镜像关联的 Play 按钮，一个 Run Image 弹窗就出现了，已准备好运行该镜像，如图 14-8 所示。

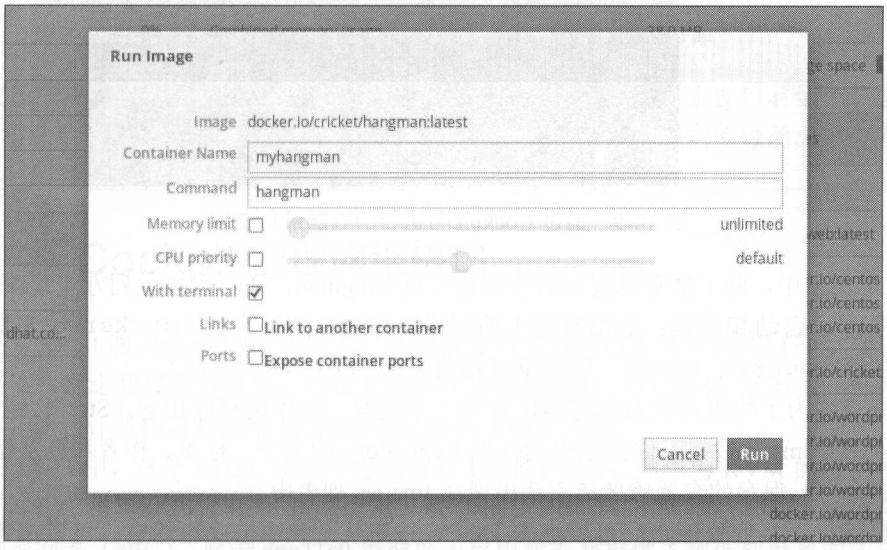


图 14-8 从 Cockpit 运行一个 Docker 镜像

图 14-8 中给出的是一个简单的例子。在这个例子中，我想从 hangman 容器中运行 hangman 命令。通过将容器命名为 myhangman，我可以认出它并在之后想要时再次启动它。

注意，这里可以选择设置容器的内存限制或 CPU 优先级。With terminal 复选框被选中，这没什么问题因为我想通过终端窗口交互式地运行 hangman 游戏。还有将容器与别的容器链接联起来或将容器的端口暴露给宿主机的选项。

如果镜像成功运行（实际上就是执行了一个 `docker run`），运行的新容器应该出现在 Containers 选项卡中——在 Containers 标题的下面。选中这个容器的名字，可以查看这个容器的信息，管理容器，以及通过终端窗口管理容器（假设是用这种方式运行的容器）。图 14-9 展示的示例是我创建的 myhangman 容器，在 Cockpit 中的一个终端窗口运行 hangman。

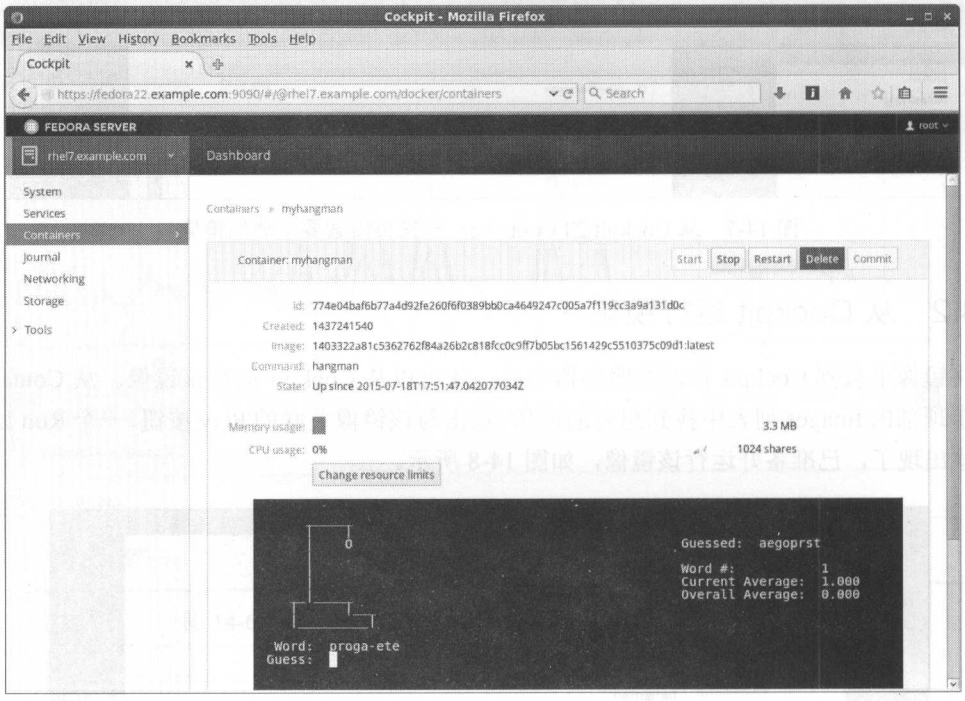


图 14-9 通过 Cockpit 与运行的容器进行交互

在这个例子中，除了能够从展示的终端窗口玩 hangman，还可以按下列方式管理容器。

- **停止、启动和重启**：容器页面上的按钮让你能够停止容器（`docker stop`）、启动容器（`docker start`）以及重启容器（`docker restart`）。
- **提交**：为了将当前状态的容器保存为一个镜像，可以选择停止它（Stop 按钮），而后点击 Commit 按钮。你会被提示提供 Repository 的名字、标签、作者和要从镜像运行的命令。保存的容器镜像接着会出现在 Images 列表中。
- **删除**：一旦容器停止，可以点击 Delete 按钮从系统中彻底删除它。可以在容器仍在运行的时候点击 Delete 按钮，但这会提示你要强制删除该容器。
- **修改资源限制**：当容器正在运行时，可以修改内存限制和 CPU 优先级。点击 Change Resource Limits 按钮，拖动代表内存和 CPU 的滑块。

如果容器正作为服务运行，终端界面可能不是必需的。通过取消勾选 With terminal，可以在不打开终端窗口与容器进行交互的情况下运行容器镜像。例如，图 14-10 展示了一个运行中的 Web 服务器容器，其并未打开终端来显示应用程序的标准输出。

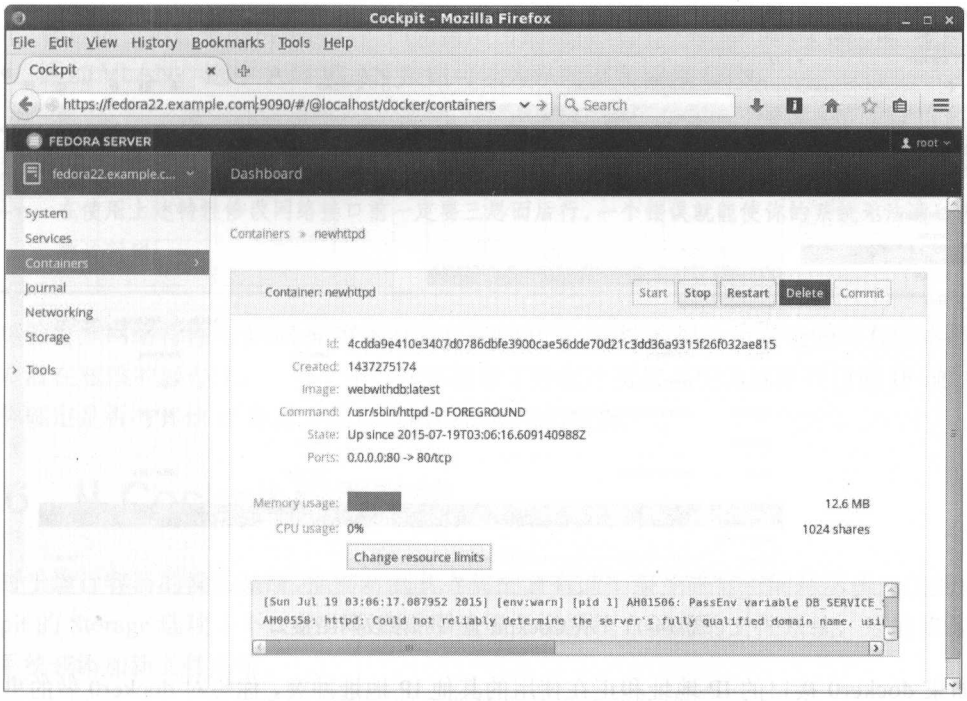


图 14-10 作为服务运行的容器可能不需要终端界面

即便没有为这个 Web 服务打开终端，仍然可以从屏幕底部的日志窗口看到来自应用程序的输出，仍然给能够查看并修改资源限制。你还能看到产生该容器的镜像的名字以及容器内的 TCP 80 端口被映射到宿主机上所有网络接口的 80 端口。

14.5 从 Cockpit 管理网络接口

通过 Cockpit 的 Networking 选项卡，可以查看和处理所选系统上的所有网络接口。随着这些网络接口发送和接收数据包，可以查看网络流量的峰值。图 14-11 展示了一台物理机的网络接口，这台计算机运行的是 Red Hat Enterprise Linux 系统。

在图 14-11 中，这个 RHEL 系统不但运行着 Docker 服务，还作为 KVM 的宿主机系统运行着（允许计算机运行虚拟机）。这些情况体现在 docker0 和 virbr0 网络接口上。这台机器还有两个物理网卡（NIC）：网卡 enp11s0 当前处于启用状态并提供了到外部世界的连接，网卡 enp12s2 此刻处于停用状态。

选中 docker0 来查看该接口的更多详细信息。从为 docker0 所展示的 Networking 界面中可以看到宿主机系统在 docker0 网络上的 IP 地址是 172.17.42.1/16。随着容器启动起来，它会被分配位于 172.17.0.0 网络中的地址。通过这个界面，能够关闭 docker0 网桥或删除它（这大概不是你想做的事情）。

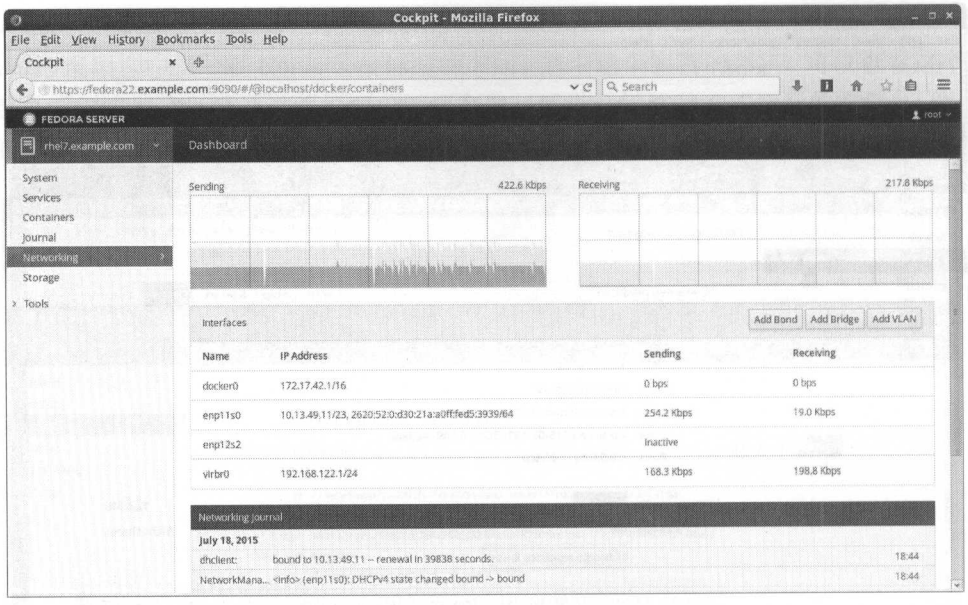


图 14-11 从 Cockpit 查看和修改网络接口

如果 docker0 接口的 IP 地址和正在使用的其他 IP 地址冲突，你想对 docker0 做的事情可能是修改该接口的 IP 地址。可以点击 Configure 按钮来进行修改，该按钮用于配置与该宿主机的 docker0 接口关联的 IPv4 地址。

选择启用的物理 NIC（enp11s0），可以看到与该接口关联的 NIC 类型（Broadcom）以及它的 MAC 地址。Status 展示了与该接口关联的 IP 地址以及该地址设置的方式（这里是通过 DHCP 设置）。Carrier 一行表明这是一个 100 Mbit/s 的 NIC，并且它在系统启动时自动连接到网络。与 docker0 接口一样，如果想修改 enp11s0 的 IP 地址（尽管在这个例子中，NIC 是从 DHCP 服务器接收 IP 地址的）就可以修改。

virbr0 网桥上的网络接口是这个例子中要关注的最后一个网络接口。默认情况下，在配置了进行 KVM 虚拟化的 RHEL 系统上，libvirtd 服务会为宿主机上的虚拟机分配 IP 地址。KVM 宿主机自己被分配的 IP 地址是 192.168.122.1，KVM 宿主机能够将 192.168.122.0/24 网络中的其他 253 个地址分配给该宿主机上运行的虚拟机。

Networking 选项卡提供了一种获取所选系统上所有网络接口的设置与活动的详细视图的方法。可以实时查看所选系统的网络活动总量，或只查看所选网络接口的网络流量，也可以在 Cockpit 中对网络接口进行一定程度的配置，具体如下。

- **添加 Bond：选中** Add Bond 按钮可以将多个网络接口与同一 IP 地址绑定在一起。可以选择绑定模式（bonding mode），如 Round Robin（网络流量会分配给与该地址绑定的所有 NIC）、Active Backup（如果主 NIC 故障，其他 NIC 准备好接管）以及其他绑定模式。
- **添加 Bridge：选中** Add Bridge 按钮可以向已有的网络接口添加网桥，以便新网桥能

够直接使用已有的网络接口，而无需路由到该接口。

- **添加 VLAN：选中** Add VLAN 按钮可以为系统添加虚拟 LAN。



注意

在使用上述特性修改网络接口前一定要三思而后行。一个错误就能使你的系统无法通过该网络被访问到。

尽管有些网络特性与 Docker 容器没有直接的联系，能够监视一个系统的所有网络接口也是查看潜在瓶颈的好办法。一瞥之下，它还展示了分配给所选系统的网络接口的 IP 地址，这能帮你确定是否与其他 IP 地址冲突以及网络已使用的范围。

14.6 从 Cockpit 配置存储

对于运行容器的操作系统，能够看到其存储配置有助于避免磁盘空间耗尽所产生的问题。Cockpit 的 Storage 选项卡不仅能让你查看所选服务器系统上创建的文件系统，还能修改这些文件系统或添加新文件系统。

图 14-12 展示了一个简单的虚拟机服务器的文件系统和其他存储信息。

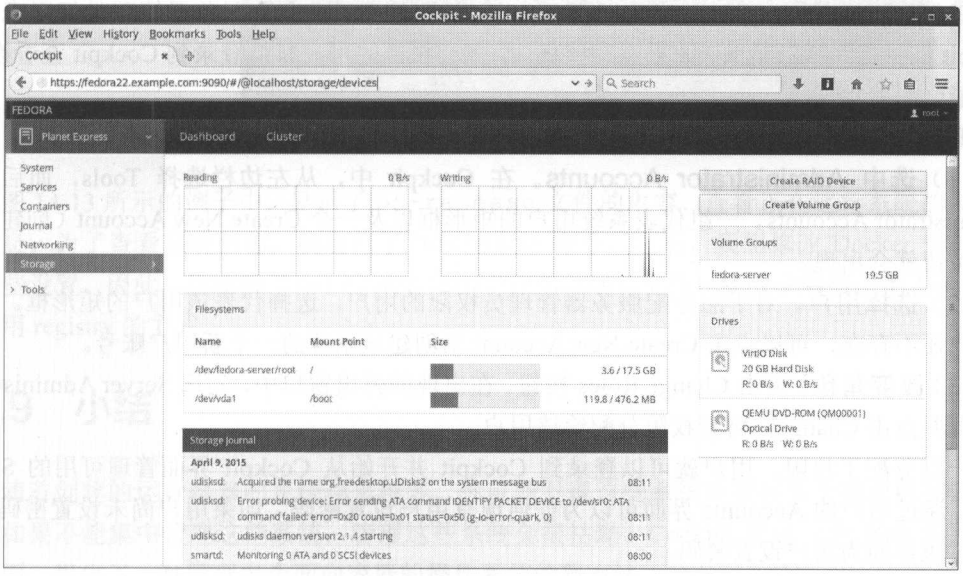


图 14-12 在 Cockpit 中展示和修改所选服务器的文件系统

在图 14-12 中，在 Drives 下可以看到有一个 20 GB 的 VirtIO Disk 为系统提供存储。该驱动器的大部分（19.5 GB）被分配给了名为 fedora-server 的 LVM 卷组。在 Filesystems 标题下，root 文件系统被配置为从`/dev/fedora-server/root` 逻辑卷获得 17.5 GB 的空间。`/boot`

文件系统位于一个单独的分区上（/dev/vda1），占用了不在 fedora-server 卷组中的剩余 500 MB 磁盘空间。

如果向正在查看的系统附加了额外的硬盘（无论是物理硬盘，还是附加网络存储，如 iSCSI），该驱动器会出现在 Drives 标题下。可以选中该磁盘，然后选择格式化它，在其上创建文件系统并挂载它。

使用 Storage 选项卡中的按钮，可以用现有磁盘空间创建特定类型的存储。选中 Create RAID Device 按钮，将两个或多个磁盘分区连接在一起，生成各类型的 RAID 设备。选中 Create Volume Group 按钮，从现有磁盘分区中创建 LVM 物理卷、卷组以及逻辑卷。

14.7 在 Cockpit 中做其他管理工作

在 Cockpit 中，通过 Tools 标题下的选项能够完成其他几项任务。如果想将服务器管理员权限授权给一个拥有所选系统账号的用户，就可以通过 Administrator Accounts 选项完成该操作。在 Tools 下，还可以打开所选系统的终端窗口或（就这个例子的 Red Hat Enterprise Linux 而言）查看该系统的订阅信息，或是将系统注册到 Red Hat。

在 Cockpit 中管理管理员账户

任何用户，只要拥有 Cockpit 所在系统的账号就可以被分配服务器管理员权限来修改 Cockpit 服务器系统的设置。如果没有该权限，拥有账号的用户只能登录到 Cockpit 查看设置，但不能修改。

要为所选用户添加服务器管理员权限，按照如下步骤操作。

（1）**选中 Administrator Accounts**。在 Cockpit 中，从左边栏选择 Tools，而后选中 Administrator Accounts。一组代表系统用户的矩形框以及一个 Create New Account（创建新账号）按钮就会出现。

（2）**选择用户**。对于想分配服务器管理员权限的用户，选择代表该用户的矩形框。如果该用户还不存在，可以点击 Create New Account 来创建该系统的一个新用户账号。

（3）**改变角色**。点击 Change Roles 按钮。在出现的弹出窗口中，勾选 Server Administrator 复选框并点击 Change，将新权限分配给该用户。

一旦分配了权限，用户就可以登录到 Cockpit 并开始从 Cockpit 界面管理可用的 Server 系统。通过用户的 Accounts 界面可以为任何现有用户设置密码，如果用户尚未设置密码，可以通过该界面为用户设置密码。

14.8 在 Cockpit 中打开终端

并不是你想用来管理容器和其他系统资源的所有功能通过 Cockpit 都可用。你也许发现自己想用 Dockerfile 构建一个容器，或者想用 Cockpit 用户界面不支持的 docker 命令选项。

在这些情况下，Cockpit 打开所选服务器的终端窗口会让事情变得容易些。

对于已添加到 Cockpit 界面的服务器，要打开到它们的终端窗口，要先选择想访问的服务器。而要选择服务器，可以从 Cockpit 窗口左上角的机器列表中选择想要的系统。接着打开 Tools 和 Terminal。

使用为该服务器分配的认证信息，Cockpit 在其界面中打开一个终端窗口并登录到该系统。图 14-13 展示的是打开的 Fedora Atomic 系统的终端窗口，这个终端窗口已运行了一些命令。

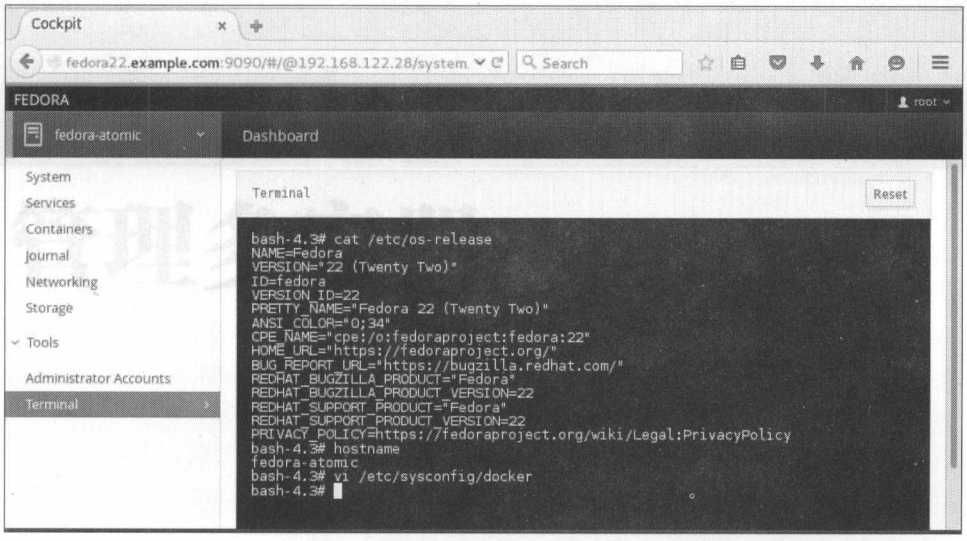


图 14-13 从 Cockpit 终端在服务器上直接运行命令

图 14-13 所示的例子中，显示了 os-release 文件的内容，查看正在运行系统的发布版本信息。为了查看系统名称，我运行了 hostname 命令。之后，我想修改 Docker 守护进程的某些设置，因此，我编辑了/etc/sysconfig/docker 文件，修改了系统上的 Docker 服务使用 registry 的方式。

14.9 小结

随着创建的运行容器的系统越来越多，你可能发现这些系统分布在多个物理机和云环境上。如果不能集中管理这些系统，管理这些系统会很枯燥。Cockpit 通过一个 Web 浏览器用户界面，提供了一种管理多个面向容器的操作系统的新方法。

尽管仍处于发展的早期阶段，现在已经可以使用 Cockpit 管理容器和镜像，以及与其相关联的底层存储和网络接口。目前，Cockpit 允许管理基于 Fedora、Red Hat Enterprise Linux 和 Atomic Host 的系统。

本章带读者逐步使用 Cockpit 来管理面向容器的操作系统以及其上运行的容器。对于使

用 Kubernetes 进行容器编排这样的 Cockpit 高级用法尚在开发中。要了解这些特性的更多信息，参见第 15 章和第 16 章，它们分别介绍了如何创建一体化的 Kubernetes 系统和集群。

到目前为止，本书专注于创建和使用单个的 Docker 容器。本书接下来的部分会带领读者实践使用 Kubernetes 和其他编排机制来管理多个相关容器。

第四部分

管理多容器

第 15 章

使用 Kubernetes 进行容器编排

本章主要内容

- 了解 Kubernetes
- 在一个系统上搭建 Kubernetes master 和 node
- 启动和管理 Kubernetes 中的 pod

多年来，像 Google、Twitter 和 eBay 这样的大型互联网公司一直在使用容器部署应用程序。虽然 Docker 容器格式为构建微服务形式的应用程序提供了一种优雅的解决方案，但它并未将部署和管理多个需要协同工作并随着需求上升而向上扩展的容器所需的一切都纳入其中。

当这些公司开始创建自己的工具来开发、部署、管理和扩展容器化应用时，他们意识到这项工作实在太巨大了，即便最大的公司也难以独自承担。然而，随着一些人构建自己私有的容器管理工具以及不同的开源项目开始用不同的方法解决问题，刚刚萌芽的容器运动受到了分裂的严重威胁。

Google 和 Kubernetes 项目加入进来。Google 将 Kubernetes 作为开源项目发布，其目标是对容器化应用的管理方法进行标准化。Kubernetes 将作为公司数据中心之上的一层管理程序，而开源项目可以致力于解决容器管理的其他方面（如创建精简的操作系统或图形化管理工具）。

2015 年 7 月，Kubernetes 达到了 1.0 里程碑，这意味着该项目已经为用于生产做好了准备。与 Kubernetes 1.0 一起，Google 还组建了云原生计算基金会（Cloud Native Computing Foundation，CNCF）。CNCF 作为与 Linux 基金会（http://collabprojects.linuxfoundation.org）合作的项目，获得了一个强有力的开始。加入 CNCF 的公司有 Red Hat、eBay、AT&T、Cisco、IBM、Intel、Twitter、Mesosphere、VMware 以及其他公司。

本章的目标是带读者领略 Kubernetes 的特性，并探讨 Kubernetes 是如何增强数据中心部署 Docker 格式容器的整体体验的。参与 Kubernetes 项目就有机会使用那些非常成功地进行容器部署的特大型公司所使用的工具。

15.1 了解 Kubernetes

创建容器为个人使用并不需要多少资源或计划，只要启动 Docker 服务并运行一些 `docker` 命令就可以了。但当需要依赖所创建的容器以一种安全、可靠、可升级并且可扩展的方式来支撑最为关键的应用程序时，新的挑战就出现了。例如，要如何处理下面这些工作。

- 部署包含多个服务（如 Web 服务器、数据库服务器和认证服务器）的应用程序，想让它们保存在各自的容器中但仍彼此交互。
- 跟踪容器提供的服务，以便其他容器知道在哪里能找到其所需的服务（例如，Web 服务器想知道为其提供数据的数据库在哪里）。
- 配置系统，以便容器崩溃或需求超过当前容器的能力时系统能够提供额外的容器。
- 集中管理一组运行在宿主机系统上的容器。
- 将容器迁移到其他宿主机系统上，以便能够停止宿主机进行维护。

Kubernetes（https://github.com/GoogleCloudPlatform/kubernetes）在许多方面为创建与管理应用程序（从研发到生产）提供了新的思维方式。下面这些方面，Kubernetes 不仅可以改变管理容器的方式，而且还能完全改变看待数据中心的方式。

- **以 DevOps 模式运维**：Kubernetes 寻求成为更大的 DevOps 运动的一部分——该运动中，应用程序研发人员和 IT 职员间的紧密沟通使持续部署成为可能，同时又不会给数据中心造成大的混乱。在 DevOps 模式中，软件开发人员要担负起更大的责任，将应用程序需要的所有软件组装到一起，而不是依赖系统管理员在宿主机系统上安装合适的组件。
- **创建公共服务**：如今，应用程序常常通过 IP 地址和端口号来请求其他应用程序提供的服务。有了 Kubernetes，就可以定义容器化的应用程序——提供其他容器可以使用的服务的应用程序。Kubernetes 处理服务提供容器和访问容器之间的连接细节。一组的务可以由一组标签来定义。
- **使宿主计算机更为通用**：宿主计算机将更为通用，而不是为运行不同的应用程序而配置不同的操作系统。因此，公司内的每个组织将不再拥有自己的要进行部署和维护的计算机。相反，用来运行一个组织的服务的物理机池与公司其他人使用的完全相同，并通过不同的标签来与其他组织的服务保持彼此分离。
- **稳定数据中心**：Kubernetes 的目标是创建一致的应用编程接口（API），以此为运行容器化应用带来稳定的环境。任何云供应商只要支持这些 API，开发人员就能够创建可以在其上运行的应用程序。这个可靠框架意味着开发人员只需了解 Kubernetes 的版本以及自己所需的服务，而无需关心数据中心的具体配置。

当然，要让这些目的变为现实所需完成的大多数开发工作尚在早期阶段，但我把目标罗列于此，是为了让读者能了解 Kubernetes 的目标以及这些目标是如何与容器驱动的数据中心这一未来相契合的。再者，我也想向读者展示如今能用 Kubernetes 做些什么。

接下来的几节将描述可以立即使用的 Kubernetes 的特性。

15.2 开始使用 Kubernetes

Docker 管理的实体被称为镜像和容器，而 Kubernetes 将这些实体包装在 pod 中。一个 pod 可以持有一个或多个容器，pod 是 Kubernetes 管理的基本单元。与直接用 Docker 管理容器相比，Kubernetes 用 pod 管理容器带来了诸多优势。

- **多 node**：Kubernetes 可以在多个 node 上部署一组 pod，而不是一台计算机上只部署一个容器。基本上，node（之前称为 minion）提供了容器运行的环境。
- **复制**：Kubernetes 可以作为 pod 的复制控制器。这意味着你可以设置任何时候有多少特定的 pod 正在运行。如果需要更多 pod 来满足你定义的复制数量，Kubernetes 可以启动更多 pod。
- **服务**："服务"这个词在许多不同的计算环境中被使用。对 Kubernetes 而言，它意味着开发人员可以为特定的 IP 地址和端口分配一个服务名（ID），而后分配一个 pod 来提供该服务。Kubernetes 在内部跟踪该服务的位置，因此 Kubernetes 知道如何将来自其他 pod 的服务请求导向正确的地址和端口。

如果选择自己搭建 Kubernetes，如接下来的两章所述，在动手前，要理解下面这些概念：

- **Kubernetes master**：Kubernetes master 是一个控制器，通过它可以部署和管理 pod、复制控制器、服务、node 以及 Kubernetes 环境的其他组件。要创建 Kubernetes master，需要设置并运行多个 systemd 服务，包括 etcd、kube-api-server、kube-controller-manager 和 kube-scheduler。本章后面会描述启动这些服务的命令。
- **Kubernetes node**：Kubernetes node（之前称为 minion）提供了容器实际运行的环境。要作为 Kubernetes node 运行，一台计算机必须设置并运行多个 systemd 服务，包括 docker、kube-proxy 和 kubelet services，这些服务要在 Kubernetes 集群中的每个 node 上运行。（本章后面会描述这些服务。）
- **`kubectl` 命令**：Kubernetes 的多数管理工作是在 master 上用 `kubectl` 命令完成的。使用 `kubectl`，可以创建、获取、描述和删除 Kubernetes 管理的任何资源（pod、复制控制器、服务等）。
- **资源文件（YAML 或 JSON）**：创建 pod、复制控制器、服务和 Kubernetes 中的其他资源时，`kubectl` 命令期待创建该资源所需的信息以两种类型的结构来描述：YAML 格式或 JSON 格式。本章后面将给出这些格式的资源文件示例。

了解 Kubernetes 如何工作的最好方法就是配置一个或多个系统运行 Kubernetes。为了演示，我创建了两种不同的 Kubernetes 配置（本章使用了一个，下一章使用了另一个）。

- **一站式 Kubernetes**：一个简单的一站式 Kubernetes 系统在同一个宿主机上运行了 master 和 node 的全部特性。这种配置能让你尝试 Kubernetes 来了解它的工作方式，但并不适用于生产环境。

- **Kubernetes 集群**：一个 Kubernetes 集群要有一个 master，还要有至少两个 node，每个 node 运行在独立的系统上。这种配置可以用于生产，并且在需要增加容量时只需要增加 node 就行。

有几种不同的方式来尝试 Kubernetes。最重要的是，要先准备一个 Linux 系统，这个系统应该一直追随着 Kubernetes 的开发。之后，可以采用下面给出的任何方式部署 Kubernetes：

- 创建虚拟机，并在虚拟机上安装 Kubernetes；
- 将 Kubernetes 直接安装在硬件上；
- 使用虚拟机部署工具，如 Vagrant（它可以在 Linux、MacOS 和 Windows 上运行 Linux 虚拟机）；
- 在云提供商上尝试 Kubernetes。



注意

参考“创建 Kubernetes 集群”页（https://github.com/GoogleCloudPlatform/kubernetes/blob/release-1.0/ docs/getting-started- guides/README.md），了解部署和使用 Kubernetes 集群的不同方法。

在本书搭建 Kubernetes 的两个例子中，操作系统使用的是 Fedora 22。虽然我在虚拟机上安装了 Fedora 22，但如果你愿意，可以在物理机上安装它。在本章中，我描述了如何搭建 master 和 node 的所有特性在同一个系统上的一体化 Kubernetes。

15.3 搭建一体化 Kubernetes

要尝试 Kubernetes 的配置——这个配置在单个系统上提供 Kubernetes master 和 node 的全部特性，我想让你安装 Fedora 22。你可以在虚拟机或物理机上进行安装。下面是要做的事情。

（1）**获取 Fedora**：下载并安装最新版的 Fedora。可以从 https://getfedora.org 获取 Fedora 的发行版。安装媒介无论是工作站（workstation）还是服务器（server）都可以很好地工作。我用的是 Fedora 22 工作站媒介，并通过该媒介进行安装。

（2）**安装 Fedora**：安装一个 Fedora 系统。Kubernetes 并不需要安装桌面界面，通过命令行就可以很好地工作。然而，我安装了一个基础服务器加桌面，这样就可以得到使用桌面工作的便利。

（3）**更新 Fedora**：为了保证有最新的软件，以 root 用户运行下面的命令：

```
# yum update
```

（4）**重启**：一旦更新完成，重启 Fedora 系统。

```
# systemctl reboot
```

Fedora 系统重启后，就准备好要开始安装 Kubernetes 了。

15.3.1 安装并启动 Kubernetes

Kubernetes 的软件包以及其他相关软件可以从 Fedora 的 basic repository 中获取到，无需任何特殊配置。然而，要获取最新版的 Kubernetes，我建议启用 update-testing repository。下面是启用该 repository 来安装 Kubernetes 以及之后开始配置它的方法。



警告

当使用 update-testing repository 获取最新版的 Kubernetes 软件时，该软件也许并不稳定。如果你发现情况的确如此，请卸载 Kubernetes，并去掉`--enablerepo=updates-testing`选项重新安装它。

（1）**启用 repository 并安装 Kubernetes**。在 Fedora 系统中，通过 update-testing repository 安装 Kubernetes：

```
# yum install -y --enablerepo=updates-testing kubernetes etcd
```

当我试着运行这个命令时，所安装的软件包有 kubernetes、kubernetes-master、kubernetes-node、kubernetes-client、etcd 以及其他几个软件包。kubernetes 软件包的版本是 1.0。

（2）**关闭 firewalld**。为了让 Kubernetes 工作，需要安装 iptables（如果没有安装就运行`yum install iptables`）。然而，因为 Kubernetes 非常依赖防火墙规则进行组件间通信，所以我建议关闭任何会冲突的防火墙管理器。运行这些命令确保防火墙功能被暂停并关闭（如果 iptables 服务在系统上被启用了，也要关闭）：

```
# systemctl stop firewalld
# systemctl disable firewalld
```

（3）**配置 master 和 node 服务**。通常，在 master 和 node 上对 Kubernetes 进行配置是告诉它们如何一起通信。这些配置文件是`/etc/etcd/etcd.conf`以及`/etc/kubernetes`目录中的文件。然而，由于 master 和 node 的大多数服务默认设置为在本地宿主机上查找彼此，因此这些信息就不需要添加到一体化系统中。

但有一个修改可以简化单个 pod 的启动，我建议你做一下。这要编辑`/etc/kubernetes/apiserver`文件，从 KUBE_ADMISSION_CONTROL 这一行删掉“ServiceAccount”。这让你在进行简单的 pod 部署时绕过 ServiceAccount 特性。下面两行代码分别展示了初始的行（已注释掉）以及去掉 ServiceAccount 后的行：

```
#KUBE_ADMISSION_CONTROL="--admission-control=NamespaceLifecycle,
NamespaceExists,LimitRanger,SecurityContextDeny,ServiceAccount,
ResourceQuota"
KUBE_ADMISSION_CONTROL="--admission-control=NamespaceLifecycle,
NamespaceExists,LimitRanger,SecurityContextDeny,ResourceQuota"
```

（4）**启动 master 服务**。要启动 Kubernetes master 需要的服务并核实服务已经运行，可以执行下面的命令：

```
# MSERVICES="etcd kube-apiserver kube-controller-manager
➥kube-scheduler"
# systemctl restart $MSERVICES
# systemctl enable $MSERVICES
# systemctl is-active $MSERVICES
active active active active
# systemctl is-enabled $MSERVICES
enabled enabled enabled enabled
```

（5）**启动 node 服务**。要启动 Kubernetes node 需要的服务并核实服务已经运行，可以运行下面的命令：

```
# NSERVICES="kube-proxy kubelet docker"
# systemctl restart $NSERVICES
# systemctl enable $NSERVICES
# systemctl is-active $NSERVICES
active active active
# systemctl is-enabled $NSERVICES
enabled enabled enabled
```

（6）**检查 Kubernetes**。运行下面这些命令来确认一体化 Kubernetes 系统已准备好发挥 node 的作用并开始运行 pod：

```
# kubectl describe node 127.0.0.1
Name: 127.0.0.1
Labels: kubernetes.io/hostname=127.0.0.1
CreationTimestamp:    Sun, 23 Aug 2015 12:05:23 -0400
...
# kubectl get node
 NAME        LABELS                                  STATUS     AGE
 127.0.0.1   kubernetes.io/hostname=127.0.0.1         Ready      9m
```

随着 master 服务和 node 服务在本地运行，现在可以启动 pod 来尝试 Kubernetes 配置了。

15.3.2 在 Kubernetes 中启动一个 pod

为了说明在 Kubernetes 中如何创建 pod，我新建了一个简单的 yaml 文件。该文件的属性如下。

- yaml 文件是 Pod 类型，这由 kind 标签指明。
- 当创建 pod 时（kubectl create 命令），它启动了一个 Web 服务器实例和一个数据库服务器实例。由于我仅运行一个 pod 实例，因此就不需要 Kubernetes 的 ReplicationController 来启动多个 pod 实例。
- 既然只定义了一个 node（本地系统），容器就在本地系统启动和运行。
- 并没有针对 Web 服务器或数据库服务器定义的 Kubernetes service。因此，要让 pod

运行的服务能够被访问到，Web 服务器暴露了本地宿主机的 TCP 80 端口，而数据库服务器暴露了本地宿主机的 3306 端口。

- 如果有容器被杀死，Kubernetes 会启动该容器的另一个实例来取代它。

按照接下来的步骤来定义 pod 并在一体化 Kubernetes 系统中创建 pod。

（1）**定义 pod 的 yaml 文件**。创建用来定义 pod 的 yaml 文件，我使用的文件名是 web-db.json。如果要复制该文件来使用这个 pod 定义，要确保维持缩进。再者，只要能连接到外网，就无需专门做任何事情来获取其包含的容器。当创建 pod 时，会从 Docker Hub 拉取这两个容器镜像。下面是该文件的内容：

```
apiVersion: v1
kind: Pod
metadata:
  name: web-db-pod
  labels:
    app: web-db-pod
spec:
  containers:
  - name: mywebdock
    image: "cricket/webdock"
    ports:
      - containerPort: 80
        hostPort: 80
  - name: mydbdock
    image: "cricket/dbdock"
    ports:
      - containerPort: 3306
        hostPort: 3306
```

apiVersion 必须与你使用的 Kubernetes 版本保持一致（这个例子中使用的是 v1）。Kubernetes 文件的“kind”是 Pod。pod 本身的名字是 web-db-pod。

这里使用了两个容器镜像，即 mywebdock（来自名为 cricket/webdoc 的镜像）和 mydbdock（来自名为 cricket/dbdock 的镜像）。mywebdock 容器将其服务通过 TCP 80 端口暴露给宿主机系统的相同端口，而 mydbdock 暴露了 TCP 3306 端口，也是暴露给宿主机上的相同端口。

（2）**创建 pod**。在一体化 Kubernetes 系统的当前目录，将刚刚描述的内容保存到名为 web-db.json 的文件中，输入下面的命令启动 pod：

```
# kubectl create -f web-db.json
pod "web-db-pod" created
```

（3）**核实 pod 正在运行**。可以使用 kubectl 命令核实 pod 的状态。在这个例子中，pod 中有两个容器，两个容器都在运行：

```
# kubectl get pod
NAME         READY     STATUS    RESTARTS   AGE
```

```
web-db-pod    2/2       Running   0          4m
```

（4）**核实容器正在运行**。使用 docker ps 命名来验证容器正在运行：

```
# docker ps
CONTAINER ID  IMAGE           COMMAND                  CREATED
  STATUS        PORTS  NAMES
eb79b0b640b8  cricket/dbdock  "/usr/bin/mysqld_saf  2 minutes ago
  Up 2 minutes          k8s_mydbdock.8c4ff0af_web-db-pod_default...
675532380c4d  cricket/webdock "/usr/sbin/httpd -D   2 minutes ago
  Up 2 minutes          k8s_mywebdock.65b9f0d7_web-db-pod_default...
b6fd829685d4  gcr.io/goog...  "/pause"              2 minutes ago
  Up 2 minutes 0.0.0.0:80->80/tcp, 0.0.0.0:3306->3306/tcp
                        k8s_POD.e80f0b79_web-db-pod_default...
```

通过 docker ps 的输出，可以看到 dbdock 容器镜像启动了 mysqld_saf 命令，并且它已经运行了两分钟。类似的，webdock 镜像正在运行 httpd 命令。Kubernetes 启动了自己的容器，如你所见，它将 TPC 80 端口和 3306 端口映射到本地系统的相同端口上。

（5）**核实应用程序正在运行**。两个端口在本地系统暴露之后，你可以使用 curl 命令查询 Web 服务器容器的 80 端口。Web 服务器的默认 Web 页面包含了一条内容为“Web server check is successful”的消息。在 Web 服务器的 cgi-bin 目录下有一个名为 action 的脚本，如果 Web 服务器能够与数据库通信，该脚本会返回消息“Docker is cool”和“DB is working”。下面是这些命令的示例：

```
# curl localhost:80
Web server check is successful
# curl http://localhost:80/cgi-bin/action

<html>
<head>
<title>My Application</title>
</head>
<body>
<h2>Docker is cool</h2>
<h2>DB is working</h2>
</body>
</html>
```

要使输出格式看上去更美观，可以在 Web 浏览器的地址栏中输入这些 URL。

15.3.3 使用 Kubernetes

当一个 pod（包含两个容器）在一体化的 Kubernetes master 和 node 上运行起来后，就可以感受 Kubernetes 是如何工作的以及如何扩展超越这个简单的示例。

要铭记在心的一件事：创建 Kubernetes 的 pod 意味着你想让该 pod 中的容器持久存在。

因此，如果 pod 中的容器被杀死了，会启动一个新容器来顶替它。示例如下：

```
# docker ps | grep dbdock
CONTAINER ID  IMAGE              COMMAND                  CREATED
  STATUS          PORTS    NAMES
eb79b0b640b8  cricket/dbdock  "/usr/bin/mysqld_saf  20 minutes ago
  Up 20 minutes          k8s_mydbdock.8c4ff0af_web-db-pod_default...
# docker kill eb79b0b640b8
34ea12d147de
# docker ps | grep dbdock
CONTAINER ID  IMAGE              COMMAND                  CREATED
  STATUS          PORTS    NAMES
a1268d5d4603  cricket/dbdock  "/usr/bin/mysqld_saf  2 seconds ago
  Up Less than a second k8s_mydbdock.8c4ff0af_web-db-pod_default_...
```

在杀死容器的几秒内，Kubernetes 自动启动了一个新容器来代替被杀死的 dbdock 容器。注意，该容器有一个新容器 ID。

如果容器出了一些问题无法再次启动，可以删除该 pod，并在再次创建 pod 之前先尝试修复该容器（或用来启动容器的镜像）。示例如下：

```
# kubectl delete pod web-db-pod
pod "web-db-pod" deleted
# kubectl get pod
NAME      READY     STATUS    RESTARTS   AGE
# docker ps
CONTAINER ID   IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
# kubectl create -f web-db.json        修复 pod 并再次创建
pod "web-db-pod" created
```

注意，在该 pod 被删除之后，再输入 kubectl get pod 命令时，那个 pod 不再出现。再者，与该 pod 关联的所有容器也被杀死。修正了 pod 或其容器之后，再次运行 kubectl create 来重建 pod。

如果 pod 中的任何容器都无法启动，删除该 pod 是一个好主意。当我首次创建自己的 Web 服务器容器时，这个容器有一个问题：一开始 pod 被创建时，这个容器可以启动，但如果容器被杀死之后，Kubernetes 尝试重启该容器会失败。如果发生这种情况，这个容器会每隔 10 秒尝试重启一次。结果是大量的错误消息被发送到日志设备，并且当输入 docker ps -a 时显示一大堆失败的容器。

15.4 小结

本章探讨了一个 Kubernetes 的简单示例，这个示例中，Kubernetes master 和 node 的特性运行在同一个系统上。在本地启动了 Kubernetes master 服务和 node 服务之后，在本地系统上部署了一个包含两个容器的 pod。随着这两个容器运行起来，可以看到这两个容器如何通过宿主机上暴露的端口来相互访问。

kubectl 是管理 Kubernetes 特性的主要命令。在本章中，我们已了解了如何使用 kubectl 创建 pod，核实 pod 正在运行，以及在完成工作后删除 pod。我们还了解了当 pod 中的容器被杀死时，Kubernetes 是如何尝试重启容器来替换它——只要 pod 仍处于活动状态。

本章介绍了 Kubernetes 的一些基本特性。然而，要了解 Kubernetes 的全部潜力，还需要扩展到本地系统之外并利用本章尚未触及的其他特性。

Kubernetes 意欲让你在 Kubernetes 集群中包含数以百计的 node。Kubernetes master 的目的就是能够启动和管理许多 pod——这些 pod 的容器分布在这些 node 之上，并使这些容器的既定数量的副本始终运行。

下一章，我会扩大本书对 Kubernetes 的覆盖面，描述在多个 node 上部署 pod、在这些 node 间创建容器的多个副本，以及利用 Kubernetes 服务简化容器发现自己到其他容器连接的方法。

第 16 章

创建 Kubernetes 集群

本章主要内容

- 理解如何使用 Kubernetes 集群
- 创建包含一个 master 和两个 node 的 Kubernetes 集群
- 在集群上部署复制控制器、服务和 pod
- 用完后，删除复制控制器、服务和 pod

要了解 Kubernetes 在部署、管理和扩展容器方面的功能范围，需要超越一体化 Kubernetes 系统（第 15 章所述），搭建 Kubernetes 集群。可以将 Kubernetes 集群作为基石，在其上建立容器管理的大型基础设施，这样的 Kubernetes 集群至少包含 3 个系统。

- **一个 master**：Kubernetes master 部署和管理容器的 pod、服务、复制控制器以及其他需要在 Kubernetes 中被管理的资源。
- **两个或更多 node**：node 是部署 pod 的地方。通过搭建至少两个 Kubernetes node，master 系统可以将工作量在这些 node 上进行负载均衡。如果一个 node 需要停机，其他 node 中的一个可以被分配来运行这些 pod。

本章致力于搭建一个包含 3 个系统的 Kubernetes 集群。在这个过程中，我会详细说明每个系统的服务和特性，然后展示这个集群的不同使用方法。

16.1 了解 Kubernetes 的高级特性

既然你已经在一体化 Kubernetes 的 master 和 node 系统上部署过 pod，是时候开始向 Kubernetes 的高级特性拓展了。当容器能够运行在具有多个 node 的 Kubernetes 集群上，Kubernetes 的特性就变得更加有用了。

当有多个 node 可以运行容器时，那些在包含了 master 和 node 的单一系统上没什么用处的 Kubernetes 特性就派上用场了。例如：

- **扩展**：随着请求增加，可以使用复制控制器为每个 pod 增加更多的副本。当这些 node 达到其容量时，可以增加更多 node，继续向上扩展。

- **跨 node 扩展服务**：Kubernetes 可以在所有 node 上跟踪服务部署的位置。当容器请求服务时，Kubernetes 能够将该请求导向到提供服务的 pod 上，而该 pod 是所有 node 上可用性最高的那个。
- **分布式存储**：使用分布式存储，容器能够在任何可以访问分布式存储的 node 上运行。

由于使用分布式存储的多数方法仍在开发中，因此本章主要关注的特性是容器使用规模的扩展以及将容器部署到多个 node 上的方法。

下一节将介绍如何搭建 Kubernetes 集群。

16.2 搭建 Kubernetes 集群

我使用 3 个虚拟机来搭建这个包含 3 个系统的 Kubernetes 集群，这些虚拟机上安装的是 Fedora 22。安装工作站（workstation）或服务器（server）都可以。一旦 Fedora 安装完成，就可以按照下面的过程在每个系统上安装软件包并启动所需的服务。下面是该过程所用的 Kubernetes、Docker 和 etcd 的版本。

- kubernetes-master-1.1.0
- kubernetes-node-1.1.0
- kubernetes-client-1.1.0
- etcd-2.0.13
- docker-1.7.1

16.2.1 第 1 步：安装 Linux

遵循以下步骤安装 Linux。

（1）**安装 Linux**。在这 3 台虚拟机上（如果你愿意的话，可以使用物理机）安装 Fedora 22。一般而言，这一过程应该也适用于其他 Linux 发行版，但有可能需要一些调整。

（2）**设置 NTP**。在 3 个系统之间保持时间同步是至关重要的。因此，在 Fedora 安装过程中选择时区时，要在这 3 个系统上启用网络时间协议。这会启动 `chronyd` 服务并与 `pool.ntp.org` 的公共时间服务器进行同步。除非你想使用自己的时间服务器，或者用 `ntpd` 替代 `chronyd` 作为时间服务，否则就不需要进一步的工作了。

（3）**设置 DNS**。确保 3 个系统全都可以通过主机名和 IP 地址相互通信。可以把每个系统添加到 DNS 服务器中（优先考虑），或者把这些系统的主机名和 IP 地址添加到每个系统的 `/etc/hosts` 文件中。如果要把每个系统添加到 `/etc/hosts` 文件中，该文件中的记录可能看起来像下面这样：

```
192.168.122.11      master
192.168.122.87      node1
192.168.122.170     node2
```

（4）**关闭防火墙**。因为 Kubernetes 大量使用 iptables 端口转发规则来帮助 pod 通信，所以应该关闭这 3 个系统上运行的任何会引起竞争的防火墙服务。在 Fedora 中，默认的防火墙服务是 firewalld。因为默认情况下会启用该服务，所以需要输入下面的命令来禁用它并停止该服务：

```
# systemctl stop firewalld
# sytemctl disable firewalld
```

16.2.2 第 2 步：搭建 Kubernetes master

登录到 Kubernetes master 系统（root 用户或拥有 sudo 权限的用户）安装 Kubernetes master 所需的软件包并设置 systemd 服务。在这一过程中，要启用并启动下列服务。

- **Kubernetes master 数据存储（etcd）服务**：etcd 的 systemd 服务保存了 Kubernetes 集群的配置数据。默认情况下，etcd 的数据被保存在/var/lib/etcd 目录中。etcd 服务有 watch 的支持，这使得 Kubernetes 组件的任何改变都会被立刻注意到并做出响应。
- **Kubernetes 调度（kube-scheduler）服务**：当 pod 尚未被安排进行部署时，kube-scheduler 服务会将未安排的 pod 绑定到可用的 node 上。
- **Kubernetes API 服务器（kube-apiserver）服务**：node 不会直接与 etcd 数据库通信，而是向 Kubernetes API 服务器服务发送请求。在更新 etcd 对象存储中的这些对象前，kube-apiserve 会先验证并处理 REST 操作。
- **Kubernetes 控制器管理器（kube-controller-manager）服务**：Kubernetes 的控制器管理器服务管理了所有 Kubernetes 集群中没有被纳入其他服务的特性。这些特性包括管理 node 的发现与监控，还有创建和更新端点控制器（endpoint controller）。

接下来的步骤描述了如何在 Kubernetes master 上安装所需的 Fedora 软件包，配置需要的服务，然后启动这些服务。

（1）**安装 Kubernetes master 软件包**。要在 Kubernetes master 上安装 kubernetes-master 和 etcd 软件包，可以输入下面的命令，：

```
# yum install -y kubernetes-master etcd
```

（2）**配置 etcd 服务**。编辑/etc/etcd/etcd.conf 文件来配置 etcd 服务，使其看上去像下面这样（其他所有行都可以被注释掉）：

```
ETCD_NAME=default
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:2379"
ETCD_LISTEN_PEER_URLS="http://localhost:2380"
ETCD_ADVERTISE_CLIENT_URLS="http://0.0.0.0:2379"
```

（3）**配置 kube-apiserver 服务**。编辑/etc/kubernetes/apiserver 文件来配置 kube-apiserver 服务。对这个文件的修改包括：

- KUBE_API_ADDRESS：修改这一行，监听所有地址（0.0.0.0），而不是只监听 localhost。
- KUBE_ETCD_SERVERS：将这个值设置为 master 服务器的位置（通过主机名或 IP 地址），紧跟着端口号 2379。
- KUBE_SERVICE_ADDRESSES：设置 Kubernetes 内部使用的地址范围。这个例子中用的是 10.254.0.0/16，以便 Kubernetes 可以使用 10.254 网络的地址。
- KUBE_ADMISSION_CONTROL：编辑这一行，删除 ServiceAccount 值和之后的逗号。删除该值使你在启动 pod 前不必配置 Kubernetes 用户。

将 KUBE_ETCD_SERVERS 中所示的 IP 地址替换为 Kubernetes master 服务器的主机名或 IP 地址。另外，可以在/etc/kubernetes/apiserver 文件中使用下面这些行（注意，最后一行有折行）：

```
KUBE_API_ADDRESS="--address=0.0.0.0"
KUBE_ETCD_SERVERS="--etcd_servers=http://192.168.122.11:2379"
KUBE_SERVICE_ADDRESSES="--portal_net=10.254.0.0/16"
KUBE_ADMISSION_CONTROL="--admission-control=NamespaceLifecycle,Names
paceExists,LimitRanger,SecurityContextDeny,ResourceQuota"
```

（4）**配置其他 Kubernetes 服务**。编辑/etc/kubernetes/config 文件，修改与不同 Kubernetes 服务相关的设置。只有 KUBE_MASTER 这行需要修改成指向 Kubernetes master 的主机名或 IP 地址（紧跟着 8080 端口）。可以保持其他设置不变。最后未被注释的行看起来应该像下面这样：

```
KUBE_LOGTOSTDERR="--logtostderr=true"
KUBE_LOG_LEVEL="--v=0"
KUBE_ALLOW_PRIV="--allow-privileged=false"
KUBE_MASTER="--master=http://master:8080"
```

（5）**启动 master 服务**。在 Kubernetes master 上，启用并启动 etcd、kube-apiserver、kube-controller-manager 和 kube-scheduler 服务。下面是实现的一种方式：

```
# MSERVICES="etcd kube-controller-manager kube-scheduler kube-
apiserver"
# systemctl restart $MSERVICES
# systemctl enable $MSERVICES
# systemctl is-enables $MSERVICES
enabled  enabled  enabled  enabled
# systemctl is-active $MSERVICES
active   active   active active
```

现在，Kubernetes master 已经准备好了。接下来，要配置两个（或更多）node。

16.2.3 第 3 步：搭建 Kubernetes node

随着 Kubernetes master 上的服务配置好并运行起来，现在可以配置 Kubernetes node 了。登录到每个 Kubernetes node（root 用户或拥有 sudo 权限的用户），安装和配置每个 Kubernetes node 运行所需的软件。在这个过程中会启用并启动下列服务。

- **Kubelet（kubelet）服务**：kubelet 服务管理着每个 node 上 pod 的方方面面，并与 master 上的 API 服务器通信。
- **Kube Proxy（kube-proxy）服务**：kube-proxy 服务管理 Kubernetes 所需的负载均衡和网络代理特性。它通过控制服务端点来做到这一点。保存在容器中的环境变量用来定义这些端点。例如，*_SERVICE_HOST 和*_SERVICE_PORT（将星号替换为某个服务名）变量定义了 IP 地址和端口号，容器可以使用它们查找服务。
- **Docker（docker）服务**：尽管不是 Kubernetes 本身的一部分，但是 docker 服务必须运行在每个 node 上，以便它最终能够管理定向到每个 node 的容器。

在每个 node 上，按照下列步骤添加 node 所需的软件，配置这些软件，并启动需要的服务。

（1）**安装 Kubernetes node 的软件包**。要将 kubernetes-node 和 docker 软件包安装到 Kubernetes node 上，可以在每个 node 上输入下面的命令，：

```
# yum install -y kubernetes-node docker
```

（2）**配置 Kubernetes proxy 服务**。kube-proxy 服务无需配置。但是，如果确实想要配置该服务，可以编辑/etc/kubernetes/proxy 文件。

（3）**配置 Kubelet 服务**。编辑/etc/kubernetes/kubelet 文件来配置 kubelet 服务。对该文件的修改包括：

- **KUBELET_ADDRESS**：修改这一行，监听所有地址（0.0.0.0），而不是只监听 localhost。
- **KUBELET_API_SERVER**：在这一行，确定 master 上的 API 服务器的位置。
- **KUBELET_ARGS**：编辑这一行，添加参数“--register-node=true”，让系统将自己注册为 master 的 node。

将 KUBE_API_SERVER 所示的 IP 地址替换为 Kubernetes master 服务器的主机名或 IP 地址。可以在/etc/kubernetes/kubelet 文件中使用下面这些行：

```
KUBELET_ADDRESS="--address=0.0.0.0"
KUBELET_API_SERVER="--api-servers=http://192.168.122.11:8080"
KUBELET_ARGS="--register-node=true"
```

（4）**配置 Kubernetes 的其他服务**。编辑/etc/kubernetes/config，修改与不同 Kubernetes 服务相关的设置。只有 KUBE_MASTER 这一行需要修改成指向 Kubernetes master 的主机名或 IP 地址（紧跟着 8080 端口）。可以保持其他设置不变。最后未被注释的行看起来应该像下面这样：

```
KUBE_LOGTOSTDERR="--logtostderr=true"
KUBE_LOG_LEVEL="--v=0"
KUBE_ALLOW_PRIV="--allow-privileged=false"
KUBE_MASTER="--master=http://master:8080"
```

（5）**启动 node 服务**。在每个 Kubernetes node 上，启用并启动 kube-proxy、kubelet 和 docker 服务。下面是实现的一种方式：

```
# NSERVICES="kube-proxy kubelet docker"
# systemctl restart $NSERVICES
# systemctl enable $NSERVICES
# systemctl is-enables $NSERVICES
enabled enabled enabled enabled
# systemctl is-active $NSERVICES
active  active  active  active
```

现在，Kubernetes 集群应该已经运行起来了。接下来可以使用 Flannel 进行网络设置。

16.2.4 第 4 步：用 Flannel 进行网络设置

可以将 Flannel 这个特性添加到 Kubernetes 集群中，用以设置 Kubernetes 内部使用的 IP 地址范围。对于本例，我将地址范围设置为 10.20.0.0/16。该范围可以让 Kubernetes 集群拥有几百个 node，每个 node 都运行几百个容器，而不会耗光 IP 地址。

本节中的步骤将在 master 和 node 上安装 flannel 软件包，将 Flannel 网络配置上传到 etcd，配置 `flanneld` 服务，然后启用并启动 `flanneld` 服务。

（1）**安装 flannel 软件包**。像下面这样，在 master 和 node 上安装 flannel 软件包：

```
# yum install flannel
```

（2）**配置 flannel**。在 master 上，打开一个文件进行编辑并给它起个像 `kube-flannel.config` 这样的名字。要让 Flannel 使用 10.20.0.0/16 这个地址范围，该文件的内容应该像下面这样：

```
{
 "Network": "10.20.0.0/16",
 "SubnetLen": 24,
 "Backend": {
   "Type": "vxlan",
   "VNI": 1
  }
}
```

（3）**将 Flannel 配置添加到 etcd**。在 master 上，输入下面的命令将 `kube-flannel.config` 文件加载到 etcd 服务：

```
# etcdctl set coreos.com/network/config < kube-flannel.config
```

（4）**设置 flanneld 服务的选项**。在 master 和 node 上，编辑/etc/sysconfig/flanneld 文件，确定 etcd 服务的位置，以便 flanneld 服务能够获取到配置信息来启动。在这个文件中，修改 FLANNEL_ETCD 这一行，确定 etcd 的主机名或 IP 地址以及要连接的端口（2379）。例如：

```
FLANNEL_ETCD="http://master:2379"
FLANNEL_ETCD_KEY="/coreos.com/network"
```

（5）**启动并启用 flanneld**。在 master 和 node 上，启动 flanneld 服务，启用该服务，并按如下方式核实其正在运行：

```
# systemctl restart flanneld
# systemctl enable flanneld
# systemctl is-active flanneld
# systemctl is-enabled flanneld
```

（6）**检查 flannel 的网络接口**。如果一切按 flannel 配置的那样运转良好，应该可以登录到 master 和 node，并看到一个 flannel.1 接口。在 node 上，还应该看到 docker0 接口在早先分配给 flannel 的同一地址范围。这里展示了 node2 上的新网络接口是什么样子：

```
# ip a | less
3: flannel.1@NONE: <BROADCAST,MULTICAST,UP,LOWER_UP>
       mtu 1450 qdisc noqueue state UNKNOWN group default
    link/ether 02:78:75:5d:71:e3 brd ff:ff:ff:ff:ff:ff
    inet 10.20.8.0/16 scope global flannel.1
       valid_lft forever preferred_lft forever
    inet6 fe80::78:75ff:fe5d:71e3/64 scope link
       valid_lft forever preferred_lft forever
4: docker0@NONE: <BROADCAST,MULTICAST,UP,LOWER_UP>
       mtu 1500 qdisc noqueue state UNKNOWN group default
    link/ether 9e:3a:23:25:68:47 brd ff:ff:ff:ff:ff:ff
    inet 10.20.8.1/24 scope global docker0
```

可以看到，flannel.1 和 docker0 接口都使用了早前分配的地址范围。如果没有看到 docker0 使用新的地址范围，可能要重启 docker 服务或重启系统。现在一切就绪，是时候开始用 Kubernetes 集群管理容器了。

16.3 启动 Kubernetes 集群中的 pod

在第 15 章中，我在一个 pod 中启动了两个容器。虽然可以这么做，但在企业环境中，预期对应用程序的访问会有涨有落，你想拥有更灵活的容器部署方式。

在本章中，我部署了两个相同的容器。我并没有将它们放在一个 pod 中，而是将它们部署在单独的 pod 中——在所谓的复制控制器中进行定义。

通过复制控制器，我不仅能够定义我想启动哪个 pod，我还能够指明我想让多少 pod 一

直运行。当创建复制控制器时，Kubernetes 尝试对这些 pod 进行负载均衡，以便它们能够在多个可用的 node 上进行部署。如果一个容器停止或被杀死，Kubernetes 会注意到并启动另一个容器来替代它。

接下来的过程描述了如何为一个数据库服务创建复制控制器，以及如何为一个 Web 服务创建复制控制器。这些复制控制器定义了它们包含的每个 pod 要一直运行的实例数量。对于每一个 pod，我还定义了一个服务，这个服务容许其他 pod 基于标签来访问它。要由 Kubernetes 跟踪提供每个服务的实际 IP 地址和端口号。

（1）**创建数据库控制器的 Kubernetes 服务**。在 master 上，我通过创建名为 `database-service.yaml` 的文件来创建代表数据库容器的 Kubernetes 服务，该文件的内容如下：

```
apiVersion: v1
kind: Service
metadata:
  name: database-service
spec:
  ports:
  - port: 3306
    targetPort: 3306
    protocol: TCP
  selector:
    app: db
```

（2）**创建 Web 服务器的 Kubernetes 服务**。在 master 上，我创建了 `web-service.yaml` 文件，将 Web 服务器 pod 设置为 Kubernetes 服务，该文件包含如下内容：

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: web
  name: webserver-service
  namespace: default
spec:
  ports:
  - port: 80
  publicIPs:
  - 192.168.122.170
  selector:
    name: web
```

（3）**创建数据库服务器的复制控制器**。在 master 上，我创建了 `database-rc.yaml` 文件来设置要启动的数据库服务器 pod 的数量。这个复制控制器的定义设置了该 pod 有两个始终运行的副本。这个 pod 启动了 `cricket/dbdock` 容器镜像，如果当 Kubernetes 要运行这个镜像时 node 上还没有这个镜像，会从 Docker Hub 拉取这个镜像。下面是该文

件包含的内容：

```
apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    name: database-controller
  name: database-controller
  namespace: default
spec:
  replicas: 2
  selector:
    selectorname: db
  template:
    metadata:
      labels:
        name: db
        selectorname: db
    spec:
      containers:
      - image: cricket/dbdock
        name: db
        ports:
        - containerPort: 3306
```

（4）**创建 Web 服务器的复制控制器**。在 master 上，我创建了名为 web-rc.yaml 的文件来设置要运行的 Web 服务器 pod 的数量。这个复制控制器的定义为名为 Web 的 pod 创建了两个副本。这个 pod 启动了名为 cricket/webdock 的容器镜像，如果 node 上还没有该镜像，就会从 Docker Hub 上拉取该镜像。这里要注意的一点是，该 pod 的“uses”定义表明它需要访问 db 服务。下面是该文件的内容：

```
apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    name: web-controller
  name: web-controller
  namespace: default
spec:
  replicas: 2
  selector:
    selectorname: web
  template:
    metadata:
      labels:
        name: web
        selectorname: web
        uses: db
    spec:
```

```
    containers:
    - image: cricket/webdock
      name: mywebserver
      ports:
- containerPort: 80
```

（5）**核实所有 node 已经准备好**。在 master 上，创建服务及复制控制器之前，核实 node 可以开始启动你定义的 pod。运行下面的命令进行核实：

```
# kubectl get node
NAME      LABELS                          STATUS  AGE
node1     kubernetes.io/hostname=node1    Ready   1d
node2     kubernetes.io/hostname=node2    Ready   1d
```

（6）**创建服务及复制控制器**。使用在当前目录中刚刚创建的服务及复制控制器文件，输入下面的命令将这些定义加载到 Kubernetes 数据库中。注意，要先加载服务，因为复制控制器需要使用这些服务。

```
# kubectl create -f database-service.yaml
# kubectl create -f web-service.yaml
# kubectl create -f database-rc.yaml
# kubectl create -f web-rc.yaml
```

（7）**审查服务、pod 和复制控制器**。如果一切顺利，应该可以在 master 上运行一些 kubectl 命令，查看刚刚创建的服务及复制控制器的结果。之后，在一个或两个 node 上运行 docker ps 命令，查看实际运行的容器：

```
# kubectl get service
NAME              CLUSTER_IP      EXTERNAL_IP PORT(S)   SELECTOR    AGE
database-service  10.254.244.240  <none>      3306/TCP  app=db      1m
kubernetes        10.254.0.1      <none>      443/TCP   <none>      15d
webserver-service 10.254.153.159  <none>      80/TCP    name=web    1m
# kubectl get rc
CONTROLLER          CONTAINER(S)  IMAGE(S)         SELECTOR
     REPLICAS   AGE
database-controller db            cricket/dbdock   selectorname=db
     2          53s
web-controller      mywebserver   cricket/webdock selectorname=web
     2          48s
# kubectl get endpoints
NAME                ENDPOINTS                       AGE
database-service    <none>                          1m
kubernetes          192.168.122.11:6443             15d
webserver-service   10.20.40.3:80,10.20.8.3:80      1m
# kubectl get pod
NAME                            READY       STATUS    RESTARTS   AGE
database-controller-3nmmd       1/1         Running   0          1m
database-controller-8lcvy       1/1         Running   0          1m
web-controller-04fi2            1/1         Running   0          1m
```

```
web-controller-5dpjb          1/1          Running   0          1m
# kubectl get endpoints
NAME                ENDPOINTS                      AGE
database-service    <none>                         2m
kubernetes          192.168.122.11:6443            15d
webserver-service   10.20.40.3:80,10.20.8.3:80     2m
# docker ps
CONTAINER ID   IMAGE              COMMAND                  CREATED
  STATUS          PORTS   NAMES
e344dcee354a   cricket/webdock   "/usr/sbin/httpd -D   5 minutes ago
  Up 5 minutes             k8s_mywebserver.be5fd5c_web-controller...
cf23d47f75a8   cricket/dbdock    "/usr/bin/mysqld_saf 6 minutes ago
  Up 5 minutes             k8s_db.78b0f953_database-controller...
0515e52602f4   gcr.io/google_containers/pause:0.8.0
                                "/pause"                 6 minutes ago
  Up 5 minutes             k8s_POD.9665f93d_database-controller...
2740ef150c6a   gcr.io/google_containers/pause:0.8.0
                                "/pause"                 6 minutes ago
  Up 6 minutes             k8s_POD.9665f93d_database-controller...
```

（8）**审查容器**。尝试从容器获取一些数据，确认它们正在运行。第一个 curl 命令显示了 Web 服务器上 index.html 文件的内容。第二个 curl 命令运行了 Web 服务器上的 action 脚本，如果成功，它会显示从数据库服务器获取的消息“Docker is cool”。你使用的 IP 地址来自名为 webserver-service 的服务的 Cluster IP 字段——kubectl service 命令的输出。

```
# curl -L 10.20.40.3:80
Web server check is successful
# curl http://localhost:80/cgi-bin/action
<html>
<head>
<title>My Application</title>
</head>
<body>
<h2>Docker is cool</h2>
<h2>DB is working</h2>
</body>
</html>
```

如果任何容器停止工作，Kubernetes 会尝试启动新容器 pod 来替代它们。

16.4 删除复制控制器、服务和 pod

当运行完前一节中搭建的复制控制器、服务和 pod 时，可以用 kubectl 命令删除它们。用 kubectl 命令删除配置的顺序在某种程度上是很关键的。通常，要按下面的顺序来运行 kubectl delete 命令：

- 删除复制控制器

- 删除服务

不需要停止容器（docker stop）或删除 pod（kubectl delete pod podname）。实际上，如果这么做，复制控制器会将它们再次启动。但是，一旦删除了复制控制器，pod 和容器就会自动停止并且 pod 会被删除。先删除复制控制器的另一个原因是，如果先删除了服务，只要 pod 对能够使用这些服务有所依赖，它们就不能工作了。

下面是删除前面示例中创建的复制控制器和服务的命令：

```
# kubectl delete rc database-controller
# kubectl delete rc web-controller
# kubectl delete service database-service
# kubectl delete service web-service
```

要核实所有东西都已删除，可以运行 kubectl get 命令。这些命令应该显示没有为该 Kubernetes 集群定义 pod、复制控制器或服务：

```
# kubectl get pod
# kubectl get rc
# kubectl get service
```

Kubernetes 集群应该回到了其过去的状态，准备部署更多 pod。

16.5 小结

Kubernetes 集群提供了在分布在一组 node 上的 pod 中部署容器的方法。通过配置 Kubernetes 集群（本章使用了一个 master 和至少两个 node），你能够看到自己使用容器的方式很容易就扩展到企业级计算。

本章的描述展示了如何使用复制控制器和服务的定义来在 Kubernetes master 和 node 系统上安装软件，如何配置实现 Kubernetes 的 systemd 服务，以及如何在 node 上运行 pod。通过这些例子，可以看到容器是如何分布在可用的 node 上的。

运行完 Kubernetes pod，本章还展示了如何用 kubectl delete 命令删除用户添加的复制控制器和服务。通过删除这些东西，它们启动的 pod 和容器会自动停止。

第五部分

开发容器

第 17 章

开发 Docker 容器

本章主要内容

- 创建容器开发环境
- 搭建 OpenShift Origin 来开发容器
- 采用良好的容器开发实践

容器中的应用程序开发可以看作是向 DevOps 软件开发模型进行迁移的一部分。容器让开发人员能够更好地控制应用程序运行所需的元素，同时无需过多考虑底层数据中心的基础设施。同样地，随着应用程序从开发到测试、从测试最终到生产，开发人员会越来越确信其应用程序将会运行。

本章会探讨开发容器化应用所需了解的事情。首先，本章展示了一些可用的开发工具，这些工具要在开始为容器开发应用程序前就位。这包括 OpenShift Origin 项目，它为容器开发提供了一个基于 Web 的用户界面。接下来，本章会过一下高效容器开发的建议以及使所生成的容器易于管理和维护的技巧。

17.1 为容器开发做好准备

在开始开发 Docker 容器前，应该考虑要准备的工具和工作环境。你可以使用预先构建好的环境开始 Docker 容器开发，但也要想想开发要如何设置。接下来的几节会涉及其中一些问题。

17.1.1 为 Red Hat 系统选择容器开发环境

要开始开发 Docker 容器，只要在 Docker 运行的系统上创建目录并在其中新建 Dockerfile 就可以了。不过，有多种方式可以搭建开发环境，以便你手边能有一整套开发工具。

有几个不同的 Docker 开发环境可以帮你迅速提高 Docker 容器开发的经验。有些情况下，你可以获得一个可启动的完整虚拟机来创建 Docker 环境；而另一些情况下，你只要下载容器，运行这些容器就能提供所需的工具集。

1. 运行 Red Hat 容器开发工具包

为了鼓励开发人员在 Red Hat Enterprise Linux 平台上构建和认证其应用程序，Red Hat 公司提供了容器开发工具包（Container Development Kit，CDK）。CDK 有多种不同形式，有针对 Red Hat 客户的，有针对合作者的，还有针对那些拥有 Red Hat 网络账号的用户的。可以通过下面的链接申请加入 Red Hat 开发者计划（这样就可以访问 CDK 和其他工具）：

```
http://developers.redhat.com
```

在使用 Red Hat 开发者计划的账号（或任何有效的 Red Hat 账号）登录后，访问下面的下载页面，从几种不同格式的 Vagrantbox 中进行选择安装：

```
https://access.redhat.com/downloads/content/293/ver=1/rhel---7/1.0.0/
x86_64/product-downloads
```

下载的 Vagrant box 可以用来安装预先配置好的 RHEL 服务器虚拟机或者 RHEL Atomic 虚拟机，这些配置是为了运行 Docker、Kubernetes 和其他对容器开发有用的软件。以这种方式使用 Vagrant 有一个巨大的优势，那就是可以在个人 Mac 或 Windows 系统上安装这些虚拟机，同时对 RHEL 容器开发环境仍拥有完全的访问。

下面是这些环境的清单，每一项都附加了对应文章的链接，这些文章描述了如何使用 Vagrant 在笔记本或桌面系统上安装 CDK 虚拟机：

- Mac OSX 上的 CDK（https://access.redhat.com/articles/1487693）；
- WIndows 上的 CDK（https://access.redhat.com/articles/1487723）；
- Fedora 上的 CDK（https://access.redhat.com/articles/1487733）。

一旦搭建好虚拟机来运行 Red Hat CDK，就可以登录到这些虚拟机，开始用它们开发和测试容器。CDK 中的大多数容器开发和部署工具都是基于文本的实用程序。

2. 运行 OpenShift

OpenShift 为开源应用程序的开发提供了一个平台，这个平台使用一组图形化工具来管理应用程序的开发、测试和生产。Red Hat OpenShift 的版本 3 整合了一些专用工具，这些工具为创建和管理 Docker 格式的 Linux 容器提供了简化的方法。

要简单快速地尝试 OpenShift，可以使用名为 OpenShift Origin 的上游项目（http://www.openshift.org/）。OpenShift Origin 网站提供了几种不同的方式来获取 OpenShift 3 平台开始试用。

- **容器内的 OpenShift**（https://github.com/openshift/origin#getting-started）：想要尽快开始使用 OpenShift，只要拉取并运行 OpenShift 容器即可。这个容器可以作为特权容器在你的系统上运行，这使你可以用这个容器从命令行或浏览器运行 OpenShift。无论用哪种界面，都可以构建容器，拉取现存容器，以及使用 Kubernetes 在 pod 中部署和管理容器。

访问 OpenShift Origin 的新手入门（Getting Started）页面，还可以找到其他资源的链接，这些资源能帮你使用对应于特定语言的 Source-to-Image（STI）镜像来创建容器。可以使用的 STI 镜像有 Ruby 的、Python 的、NodeJS 的、PHP 的和 Perl 的。

稍后会描述通过 OpenShift Origin 尝试容器内 OpenShift 的步骤。

- **Ansible 的 OpenShift**（https://github.com/openshift/openshift-ansible）：使用 Ansible 安装程序，可以在本地虚拟机、Fedora 或 Mac OSX 上搭建容器的 Kubernetes 集群。
- **云供应商的 OpenShift**：有在 Amazon Web Services（https://github.com/openshift/openshift-ansible/blob/master/README_AWS.md）或 Google Cloud Engine（https://github.com/openshift/openshift-ansible/blob/master/README_GCE.md）内运行 OpenShift 的安装说明。

Red Hat Enterprise Linux（RHEL）、RHEL Atomic Enterprise 和 RHEL Atomic Host 系统都有本地运行的 OpenShift。因此，在尝试了 OpenShift Origin 之后，你可以在其中一种 Red Hat 系统上配置 OpenShift 的企业级版本。

下列步骤将帮你在一个系统上把容器内的 OpenShift Origin 运行起来。关于 OpenShift Origin 安装和运行的详细信息，请参考项目网站上的 OpenShift 应用平台入门指南（https://github.com/openshift/origin#getting-started）。

（1）**安装 Linux**。开始安装一个新的 Linux。我在一台裸机上安装了 Fedora 22 工作站（尽管虚拟机可能更好）。对 Fedora 而言，无需特别的 repository 设置。不过，如果是在 RHEL 上执行该步骤，你需要按如下方式订阅系统并启用其他可选的 repository：

```
# subscription-manager register --auto
# subscription-manager repos --disable=""
# subscription-manager repos \
  --enable="rhel-7-server-rpms" \
  --enable="rhel-7-server-extras-rpms" \
  --enable="rhel-7-server-optional-rpms" \
  --enable="rhel-server-7-ose-beta-rpms"
```

（2）**安装容器软件**。按如下方式安装将系统作为 Kubernetes master 和 node 运行所需的软件：

```
# yum install docker kubernetes etcd
```

（3）**配置 Kubernetes**。按第 16 章所述的方式配置 Kubernetes 集群。

（4）**禁用 SELinux**。如果收到 AVC 拒绝消息，要么修正这个问题，要么禁用 SELinux。要将 SELinux 置于 permissive 模式，打开/etc/sysconfig/selinux 文件，按如下方式修改 SELINUX=行：

```
SELINUX=permissive
```

（5）**信任 Docker Registry 服务**。编辑/etc/sysconfig/docker 文件，让 Docker

服务信任之后启动的 docker registry 服务。要达成这一点，去掉 INSECURE_ REGISTRY 所在行的注释，并按如下方式进行修改：

```
INSECURE_REGISTRY='--insecure-registry 172.30.0.0/16'
```

（6）**启动 Docker 服务**。确保 docker 服务已启用并被启动：

```
# systemctl enable docker
# systemctl restart docker
```

（7）**启动 OpenShift Origin**。Docker 运行后，可以输入下面的命令拉取和运行 openshift/origin 容器。这将使容器以守护进程的方式在后台运行：

```
# docker run -d --name "origin" \
 --privileged --net=host \
 -v /:/rootfs:ro -v /var/run:/var/run:rw \
 -v /sys:/sys:ro -v /var/lib/docker:/var/lib/docker:rw \
 -v \
 /var/lib/openshift/openshift.local.volumes:/var/lib/openshift/
openshift.local.volumes \
 openshift/origin start
```

（8）**使用 OpenShift Origin 容器**。origin 容器在后台运行后，就可以执行 `docker exec` 命令打开运行容器内的 bash shell 并开始使用 OpenShift Origin。输入下面的命令来完成上述工作：

```
# docker exec -it origin bash
[openshift]#
```

（9）**运行一些 OpenShift 命令**。从 origin 容器内，可以运行一些命令来启动 registry、登录到（没有启用认证系统，因此输入任何用户名和密码都会创建一个账户），以及开始新项目：

```
# oadm registry \
  --credentials=./openshift.local.config/master/openshift-registry.
kubeconfig
# oc login
Username: joe
Password: joe
# oc new-project joeproject
Now using project "joeproject" on server "https://10.13.49.45:8443".
# oc new-app -f \
https://raw.githubusercontent.com/openshift/origin/master/examples/
sample-app/application-template-stibuild.json
...
Service "frontend" created at
    172.30.8.90 with port mappings 5432->8080.
Build "ruby-sample-build" created and started -
    you can run 'oc status' to check the progress.
```

```
Service "database" created at
    172.30.40.239 with port mappings 5434->3306.
Run 'oc status' to view your app.
```

一旦容器镜像完成构建，它们就会被部署，你就可以进行测试以确保它们能够工作。

（10）**查看 OpenShift 的 Web 用户界面**。要检查正在构建的 docker 镜像的状态并查看其进展情况，可以使用 OpenShift 的 Web 用户界面。打开本地系统的 Web 浏览器，访问下面的 URL，登录到（使用刚刚创建的用户名和密码），接受非安全连接（自签署证书），而后开始使用 OpenShift：

```
https://localhost:8443
```

（11）**签出项目**。如图 17-1 所示，登录到 OpenShift Origin 的 Web 用户界面后，通过命令行添加的项目（这里是 joeproject）应该会被显示。从这个项目的 Overview 标签页，可以查看 Apache Web 服务器的服务前端和创建的数据库服务。选中 Browse 标签页并选择 Pods，查看 pods 的构建进度。

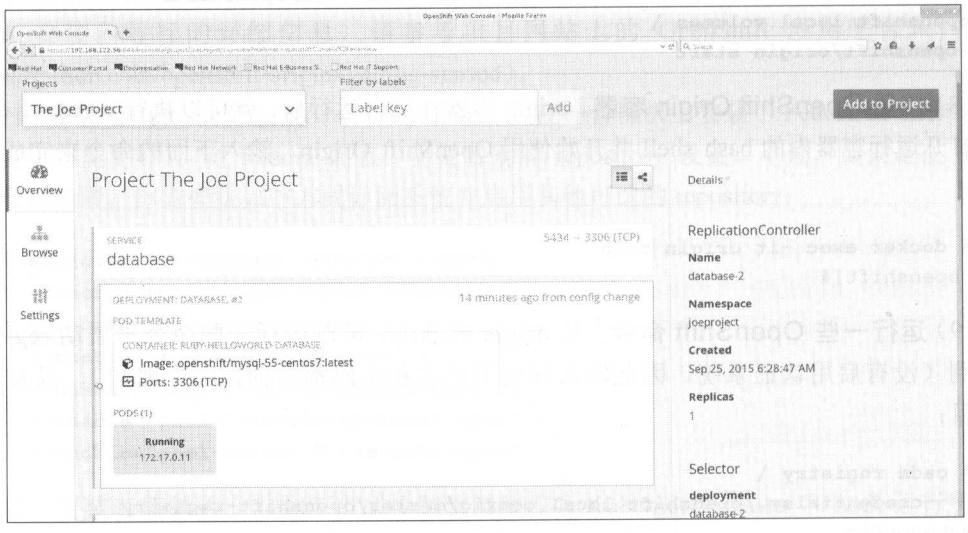


图 17-1 通过 OpenShift 的 Web 用户界面管理容器项目

一旦 pods 完成构建并且容器完成部署，就可以尝试访问 Web 服务器来确认其已经工作。如果构建项目过程中遇到任何问题，请参考 OpenShift Origin 的故障诊断指南（https://github.com/ openshift/origin/blob/master/docs/debugging-openshift.md）。

17.1.2 来自 Docker 的容器开发环境

Docker 项目自己也提供了容器开发工具，你可能会感兴趣。其中一些工具给了你构建和管理 Docker 容器的新选择——你可以使用图形化前端而不是使用命令行工具。这些工具包括

以下几个。

- Docker Kitematic（www.docker.com/docker-kitematic）：Kitematic 为 docker 提供了一个图形化前端。使用 Kitematic，无需依赖 Linux 命令行就可以构建和运行容器。
- Docker Toolbox（www.docker.com/toolbox）：Docker Toolbox 提供了一个安装程序，它使你能够在台式机或笔记本上搭建 Docker。Mac 和 Windows 也有 Docker Toolbox。使用 VirtualBox 在你自己的计算机上创建 Linux 环境，并用 Docker Toolbox 拉取其他 docker 工具（如 Docker Client、Docker Machine 和 Docker Compose）来帮你在自己的系统上使用 Docker。
- Docker Machine（www.docker.com/docker-machine）：Docker Machine 提供了一种向你自己的计算机、云供应商或你的数据中心配置 Docker 的方法。使用 Docker Machine，可以供应宿主机并在这些系统上部署 Docker Engine。最后，配置 Docker 客户端来使用该 Docker Engine。

要构建更大的容器开发环境，可以考虑下面这些来自 Docker 的工具。

- Docker Compose（www.docker.com/docker-compose）：Docker 项目提供了 Docker Compose 来帮助开发人员将容器组成更大的应用程序。
- Docker Swarm（https://docs.docker.com/swarm）：Docker Swam 为扩展 Docker 运行多个容器和多个宿主机提供了工具。

这些来自 Docker 项目的工具能够帮你搭建环境来使用 Docker，也能够让你对 Docker 的使用突破单容器应用程序。与使用命令行界面相比，如果你更习惯使用基于 Web 的界面，这里描述的许多 Docker 工具也可以让 Docker 的使用变得更容易。

17.2 采用良好的开发实践

尽管构建 Docker 镜像十分简单，还是有一些可用的技巧可以让容器开发更为高效。下面描述其中的一些技巧。

17.2.1 将文件包含到构建中或排除出去

我已经探讨过，通常最好是创建一个新目录来包含 Dockerfile 以及构建所需的其他文件。由于构建过程中该目录结构下的所有文件都会直接发送给 Docker 守护进程并复制到 Docker 维护的文件系统，因此最好使该目录下的文件数量保持最小。

有时候你也许想在保存 Dockerfile 的目录结构中存放一些其他的文件，而又不想在构建容器时包含这些文件，或者至少不是每次构建时都包含这些文件。例如，你也许想使用相同的 Dockerfile 文件来构建包含不同配置文件的容器镜像，或者你想创建一个不包含文档的更为精简的镜像。

使用 Dockerfile 所在目录的`.dockerignore`文件，可以排除构建不想包含的文件。在该文件中，使用文本和特殊字符来告知要排除什么文件。Dockerfile 参考页有该文件的描述

(https://docs.docker.com/reference/builder/#dockerignore-file)。下面是一个构建目录的内容示例，你也许想在一次 docker build 中从该目录排除一些文件：

```
# ls ~/mybuild
. .. Dockerfile .dockerignore testbuild1/ testbuild2/ README
```

每次执行 docker build 时，都可以修改.dockerignore 文件来忽略不同的文件或目录。下面的例子是排除不同文件和目录时，可以在该文件中包含的条目：

```
testbuild1
testbuild?
testbuild[12]

*
!README
```

你可能不会在同一个.dockerignore 文件中使用所有这些行，但也许会单独使用它们。testbuild1 这一行的结果是 README 文件和 testbuild2 目录下的所有文件都包含在构建中。testbuild?会排除 testbuild1 和 testbuild2 目录，只留下 README 文件包含在构建中。如果 testbuild?这样的目录有多个，testbuild[12]将明确只排除 testbuild1 和 testbuild2 目录，而不会排除 testbuild3、testbuild4 等目录。

如果你在该文件中仅放置一个星号(*)，构建将不会包含构建目录中的任何文件。然而，如果使用星号排除文件后你想把一些文件再包含进来，叹号(!)是解决这一问题的方法。在这个例子中，!README 会将 README 文件包含进来。

记住，构建中包含或排除文件并不能让这些文件包含在你正在构建的镜像中。要让镜像包含文件，你要在 Dockerfile 中使用 ADD 和 COPY 指令。排除文件只是让你无需浪费时间等待将不需要的文件复制给 Docker 守护进程以及在 Docker 目录结构中消耗不必要的存储空间。

17.2.2 利用分层

Docker 镜像是由基础镜像以及其上按层叠加的其他文件系统创建的。用来达成这一点的文件系统称为联合文件系统(union file system)。了解这些层的工作方式将有助于在运行 docker build 时管理这些层的使用方式。

高效地使用层能对你与 Docker 交互的各个层面起到帮助作用。如果数据中心的每个容器镜像都使用了相同的基础镜像，你只需在容器运行的每个节点上保留一份基础镜像。当你拉取镜像完成一个具体工作时，只需下载其他层。

通过在逻辑上将层建立在彼此之上，可以进一步提高效率。例如，可能你增加的层包含库和运行时环境所需的其他组件(如 JBoss 和 Ruby)，那么许多需要相同环境的应用程序都可以使用这个容器。因此，你在运行时环境之上添加的每个服务都会生成一个新容器，而这个容器的拉取会非常高效，因为它只需要将应用层添加到运行时环境。

当真的要构建容器时，谨记构建过程每次运行一个指令时，它都会创建一个新的层。一些在 Dockerfile 中使用指令的方法可以造成更多或更少的层被创建。

在 RUN 指令所在行放置多条命令，其结果只会为该 RUN 指令中的多条命令创建一个层。下面是这样做的一些影响：

- 在一个 RUN 指令中包含几条命令有助于在镜像内的中间层之间设置良好的边界。例如，包含了 `yum -y update && yum clean all` 的 RUN 指令行会更新容器内所有软件包，而后清理 yum 留下的缓存数据。如果接下来的构建指令失败了，在修正问题之后，下一次 `docker build` 命令可以从一个已升级并完成清理的容器开始——这个容器正是我刚刚描述的 RUN 指令的结果。
- 不利的一面是，有些情况下包含太多命令的 RUN 指令会效率低下。如果 RUN 指令的几条命令中的一个命令失败了，就不得不返回去重新执行整条指令。如果这些命令在单独的 RUN 指令行中，就只要从失败的那条命令开始就好了。

容器镜像一经创建，就可以考虑尝试将层压到一起。已经有工具被开发出来帮你做这件事。这能减少要管理的层，但它也会导致一些低效。例如，如果容器有多个层，这些层可能会被更高效地拉取到你的系统中，因为可以同时下载。再者，如同之前提到的，如果拉取时有些层已经在系统中，就不必再拉取它们。但如果层没有分开，任何修改都需要再次拉取整个镜像。

17.2.3 管理构建中的软件包

将随便什么地方获取的软件放入容器镜像可不是个好主意。从互联网随便什么网站获取的软件，最糟糕的情况下可能包含恶意软件，而不那么糟的情况下也无法保证它可以与容器中的其他软件很好地一起工作。

在选择要放入容器的软件包时，下面是要考虑的一些事情。

- **选择一个值得信赖的 Linux**：许多知名的 Linux 发行版都提供了基础镜像，这些基础镜像包含了基于它们的发行版创建容器所需的最少组件。通常，这些基础镜像也做了配置以便你可以立即在容器内安装更多软件。如果使用 Fedora、Ubuntu、CentOS、Red Hat Enterprise Linux 或其他流行的 Linux 发行版，一开始先用该发行版的官方基础镜像，这为你提供了一个值得信赖的起点。
- **移除不需要的软件包**：如果不需要基础镜像中的某个文本编辑器、诊断工具或其他软件，只要从基础镜像中移除其所包含的软件包并生成新的镜像即可。移除每个软件包都可以节省在升级和管理镜像上花费的时间。它还使镜像更小，从而使存储和移动更高效。
- **不要安装运行时不需要的软件包**：避免包含用于容器调查和问题诊断的软件，因为这些软件可以在需要时通过其他方式添加（例如，把一个工具容器添加到系统中，或者暂时将软件包安装到运行的容器中，之后再移除它）。通常而言，你应该不想在自己构建的容器中包含临时性的组件。

- **排除部分软件包**：有些包管理系统提供了防止安装特定软件的方法，你或许不想这样做。在 Fedora、Red Hat Enterprise Linux 或 CentOS 中安装软件包时，`yum filter` 命令使你能够过滤掉特定类型的软件。例如，如果运行下面的 `yum fs` 命令，`docker build` 过程中运行的任何 `yum install` 命令都会排除所安装软件包中的文档文件：

```
RUN yum fs filter documentation
```

17.3 进一步了解容器构建

随着人们对开发和使用容器的最佳方式更为了解，更多信息将会出现。我建议密切关注下面这些地方以获取对容器开发有帮助的信息。

- **回顾 Dockerfile 书写的最佳实践**（https://docs.docker.com/articles/dockerfile_best-practices）：Docker 项目提出了一系列所谓的最佳实践。这些最佳实践就如何最好地使用 Dockerfile 中的每条指令给出了具体建议。
- **查阅 Dockerfile 参考手册**（https://docs.docker.com/reference/builder）：Dockerfile 参考手册为指令以及 Dockerfile 中使用这些指令的方法提供了广泛的参考资料。如果你尚未确切地了解一条具体指令的使用方法，那么这就是你要去的地方。
- **Docker 官方镜像项目**（https://github.com/docker-library/official-images）：从 Docker 官方镜像主页查询创建和维护官方 Docker 镜像的指导。它不仅探讨了如何制作高效、可重用的镜像，还探讨了如何制作安全的镜像。

17.4 小结

作为想开发 Docker 格式容器化应用的人，你会想在手边准备一套良好的开发工具，这些工具会将你从创建独立的小容器提升至创建大型企业级应用。像 OpenShift 这样的开源项目提供了在 pod 中创建和管理可扩展容器的方法，其后端使用 Kubernetes。

可以自如地使用这套开发工具之后，本章的第二节带你了解了适用于大多数容器开发的建议。这包括来自 Docker 项目的最佳实践，以及会使 Docker 容器开发更高效、更安全的其他想法。

第 18 章

探索 Dockerfile 文件示例

本章主要内容

- 查看 Linux 发行版的 Dockerfile
- 了解来自开源项目的 Dockerfile
- 研究个人应用程序的 Dockerfile

事实上，许多人会将他们的 Dockerfile（在 GitHub 上以及其他地方）发布出来，让你有机会一览人们在工作和学习中是如何创建 Docker 镜像的。为了说明不同类型的 Dockerfile，这里我按特性挑选了一些 Dockerfile 并将它们分成几类。

- **Linux 发行版**：许多 Linux 发行版公布了生成其发行版 Docker 官方镜像所使用的 Dockerfile。查看这些 Dockerfile，以及它们有时包含的 tarball 文件，能够了解这些镜像是如何创建的。
- **开源项目**：有些开源应用项目通过创建容器镜像来为其应用程序的启动提供一种简单的方法。用来创建这些镜像的 Dockerfile 通常包含了环境变量并指定了要执行的命令，以便在运行这些应用时帮你配置这些应用程序。
- **个人应用程序**：许多人创建容器仅仅是为了在自己的台式机或个人服务器上运行想要的应用程序时能有简便的方法。有时这些容器是特定于某个具体 Linux 发行版的，因为它们需要使用特定的声音、视频或其他桌面特性，以便在本地系统上高效地工作。

本章中，我将展示组织或个人发布的一些 Dockerfile，以便读者能够了解他们用来构建 Docker 镜像的技巧。尽管这些技巧对创建 Dockerfile 而言不一定是最佳实践（从 Docker.com 上了解这方面的最佳实践指南），但通过了解其他人是如何解决问题的——你创建容器化应用时也可能会遇到这些问题，能够受到一定的启发。



警告

本章描述的示例 Dockerfile 只是为了展示人们用来容器化其 Linux 发行版或应用程序的技巧。如同运行任何软件所应做的，我建议你在使用这些 Dockerfile 构建和运行自己的容器镜像前，先调查一下纳入这些 Dockerfile 的组件的安全情况。

18.1 细察官方 Docker 镜像的 Dockerfile

在容器之前，要想运行 Linux 应用程序，通常要先搭建 Linux 系统，而后安装并配置该应用程序，使其在这个系统上运行起来。有了容器，相同的概念仍旧适用。要在容器中运行应用程序，通常要基于特定的 Linux 发行版选择一个容器，而后从该发行版添加软件（很可能是你自己的软件），为运行这个软件准备一个新的容器镜像。

本节描述的 Dockerfile 展示了 CentOS 和 Busybox 这样的 Linux 发行版创建其基础容器的方法。许多这样的容器是由 `scratch` 容器（可以从 Docker Hub 上获取）起步，而后添加了那些发行版认为最基础的软件——容器内大多数应用程序运行所需的软件。

18.1.1 查看 CentOS 的 Dockerfile

CentOS 是一个流行的 Linux 发行版，它适用于为企业使用而运行但又无需支持和升级方面保障的 Linux 系统。CentOS 构建自 Red Hat Enterprise Linux 的源代码，并提供了一种方法来试验那些通过 Red Hat Enterprise Linux 订阅才能得到的相同软件。

起初，官方的 CentOS 容器镜像并没有包含 CentOS 发行版附带的 init 系统（默认是 systemd）来启动和管理容器内的应用程序。最近，他们向 CentOS 基础镜像添加了一个特殊版本的 systemd 软件（通过 systemd-container 和 systemd-container-libs 软件包）。现在可以在 CentOS 容器内安装那些已经配置了要与 systemd 一起才能使用的服务（如 httpd Web 服务），而后通过 `systemctl` 命令启动和管理这些服务。

接下里的一节会描述用来构建 CentOS 基础镜像的 Dockerfile。之后，我会介绍如何使用该基础镜像安装 httpd Web 服务器，以及如何使用 CentOS 基础镜像将 httpd Web 服务器作为 systemd 服务进行启动。

1. 使用 CentOS 基础镜像的 Dockerfile

从 Docker Hub 可以获取 CentOS 项目的官方 Docker 镜像（https://hub.docker.com/_/centos/）。在这个页面，选择你感兴趣的版本的 Dockerfile（如 centos7），查看该 Dockerfile 的内容。接下来，转到包含该 Dockerfile 的目录（选择 `docker` 目录）寻找该 Dockerfile 包含的 tarball 文件。

下面是构建 CentOS 7 的官方 CentOS 镜像所使用的最新 Dockerfile：

```
FROM scratch
MAINTAINER The CentOS Project <cloud-ops@centos.org> - ami_creator
ADD centos-7-20150616_1752-docker.tar.xz /
# Volumes for systemd
# VOLUME ["/run", "/tmp"]
# Environment for systemd
# ENV container=docker
```

```
# For systemd usage this changes to /usr/sbin/init
# Keeping it as /bin/bash for compatibility with previous
CMD ["/bin/bash"]
```

有关这个 Dockerfile 的内容，这里有一些事情需要注意。

- 从 Docker Hub 拉取的 `scratch` 镜像（https://hub.docker.com/_/scratch）提供了底层镜像（在 FROM 这行），而 CentOS 基础镜像构建在该底层之上。在 Docker Hub 中，`scratch` 镜像被描述为用于构建 Linux 发行版的基础镜像或者是构建“超小镜像”。其他主要 Linux 发行版也是以 `scratch` 镜像起步的。
- MAINTAINER 行将 CentOS 项目罗列为该镜像的维护者。如果对该镜像有疑问，这里提供了一个邮件地址，以便联系该 Dockerfile 及其镜像的维护者。
- ADD 行会将一个 tarball 文件——其包含了 CentOS 容器的完整内容——展开并放置到容器文件系统的根（/）目录。像之前提到过的，该 tarball 文件与 Dockerfile 在 GitHub 的同一目录中。
- Dockerfile 的最后一行是 CMD 指令，默认情况下，它会在运行容器时打开 bash shell。

要查看该 tar 文件是如何构建的，去 GibHub 的 sig-cloud-instance-build 页面（https://github.com/CentOS/sig-cloud-instance-build/tree/master/docker）选择一个 Kickstart 文件。这些 Kickstart 文件与标准的 CentOS 安装程序一同用于安装最小的 CentOS 系统。每个安装的系统都能够被复制到一个用于构建与 CentOS 版本相对应的容器的 tarball 文件中。

如果研究了这些 Kickstart 文件，你会发现只是安装了最少的软件包。有一个不属于标准 CentOS 发行版的软件包被添加进来，这就是 systemd-container 软件包。该软件包含有启动和停止 systemd 服务所需的组件，但并不包含只适用于宿主机系统的那些组件（如整个改变宿主机系统状态的那些 target）。

一些自动安装的软件包被移除或排除了，如 firmware 软件包、GRUB2 bootloader、CentOS 日志和防火墙软件（firewalld 软件包）。为了让基础镜像保持很小，Kickstart 文件还屏蔽了所有软件包中文档，并移除了已经知道不需要的大文件。每个 Kickstart 所产生的 tarball 文件（不同的 CentOS 发布有不同的 Kickstart 文件）为在 CentOS 容器内运行应用程序提供了一个可以使用的简洁基础。

作为使用 CentOS 基础容器的人，你要创建自己的 Dockerfile 并将 CentOS 基础容器添加到 FROM 行。Dockerfile 的其余部分可以添加软件包（`yum install` 任何软件包）以及你自己的软件和配置文件。接下来的小节描述了如何将 httpd 软件包添加到 CentOS 基础系统容器中，而后以 systemd 服务的方式启动该 Web 服务器。

2. 向 CentOS 的 Dockerfile 添加 systemd 服务

要在基于 CentOS 镜像所构建的 Dockerfile 中运行启用了 systemd 的应用程序，只要在这个新 Dockerfile 中安装包含该服务的软件包并在容器运行时用 `systemctl` 命令启动该服务即可。下面是实现这一点的 Dockerfile 示例：

```
FROM centos:7
RUN yum -y install httpd; yum clean all; \
systemctl enable httpd.service
EXPOSE 80
RUN echo "Web Server Works!" > /var/www/html/index.html
CMD ["/usr/sbin/init"]
```

要将这个 Dockerfile 构建成一个运行 httpd 服务器的镜像，可以输入下面的命令：

```
# docker build -t systemd-httpd .
```

systemd-httpd 镜像创建成功后，可以使用下面的命令运行该镜像：

```
# docker run --privileged -d -v /sys/fs/cgroup:/sys/fs/cgroup:ro \
   -p 80:80 systemd-httpd
```

刚刚展示的命令运行了 httpd 的 systemd 服务。因为 Dockerfile 已设置启用 httpd 的 systemd 服务，所以需要将/sys/fs/cgroup 目录从宿主机挂载到容器内部。-p 80:80 选项将容器的 TCP 80 端口开放给宿主机的相同端口，而-d 选项让容器在后台运行。要核实服务正在运行，输入下面的命令：

```
# curl http://localhost
Web Server Works!
```

可以将与此相同的 Dockerfile 作为一种模型，用于启动那些在其软件包中含有 systemd 单元文件的服务。

18.1.2 查看 busybox 的 Dockerfile

busybox 是一个微型操作系统，其常用于在嵌入式系统中运行简单的可执行程序。你可以在 Docker Hub 找到 busybox 的镜像（https://hub.docker.com/_busybox/）。busybox 的 Dockerfile 引入了一个很小的 tarball 文件（只有大概 2.7 MB），这个 tarball 文件仅包含足以运行静态编译二进制文件所需的实用工具。

像 CentOS 基础镜像的 Dockerfile 一样，用来构建 busybox 自身的 Dockerfile 也始于 scratch 镜像并添加了一个 tarball 文件。下面是从 Docker Hub 获得的一个 busybox 的 Dockerfile 示例：

```
FROM scratch
MAINTAINER Jerome Petazzoni <jerome@docker.com>
ADD rootfs.tar /
CMD ["/bin/sh"]
```

要创建 Docker 镜像来运行自己的静态编译二进制文件，你可以创建自己的 Dockerfile，该 Dockerfile 包含 busybox 基础镜像并添加你的二进制文件。下面是来自 Docker Hub 上的 busybox 页面的示例：

```
FROM busybox
COPY ./my-static-binary /my-static-binary
CMD ["/my-static-binary"]
```

你得编译自己的二进制文件，因为 busybox 镜像不包含编译所需的工具。一旦编译了二进制文件，就可以将该二进制文件放在 Dockerfile 所在的目录（将 `my-static-binary` 替换为你的二进制文件名）。当运行 `docker build` 时，新镜像将包含该二进制文件并设置当容器运行的时候默认执行该二进制文件。

18.2 细察来自开源项目的 Dockerfile

越来越多的开源项目创建了容器镜像来运行它们的软件，并让这些镜像可以通过 Docker Hub 获取到。这里描述的这些项目的 Dockerfile 不同于基础操作系统容器镜像的 Dockerfile。这些 Dockerfile 是要创建直接运行的镜像，而不是用它们构建其他镜像。

看一看如何为 WordPress 和 MySQL 创建 Dockerfile 能够对如何创建镜像运行你自己的应用程序有所启发。这两个例子反映的是在后台运行的守护进程，在其运行之处提供服务。当使用这些镜像时，它们为你提供了修改环境变量的方法，通过修改环境变量可以操纵这些镜像运行的时候的行为。

18.2.1 查看 WordPress 的 Dockerfile

WordPress 是一个流行的开源内容管理系统和博客工具。Docker Hub 上有 WordPress 的官方镜像（https://hub.docker.com/_/wordpress/），还有 Dockerfile 和相关软件供你查看，或者用于构建你自己的 WordPress 容器镜像。

从 Docker Hub 的 WordPress 页面选择一个 Dockerfile 的链接。下面的示例是这些 Dockerfile 中的一个：

```
FROM php:5.6-apache
RUN a2enmod rewrite
# install the PHP extensions we need
RUN apt-get update && apt-get install -y \
  libpng12-dev libjpeg-dev && rm -rf /var/lib/apt/lists/* \
  && docker-php-ext-configure gd --with-png-dir=/usr \
  --with-jpeg-dir=/usr && docker-php-ext-install gd
RUN docker-php-ext-install mysqli
VOLUME /var/www/html
ENV WORDPRESS_VERSION 4.3
ENV WORDPRESS_SHA1 1e9046b584d4eaebac9e1f7292ca7003bfc8ffd7
# upstream tarballs include ./wordpress/
# so this gives us /usr/src/wordpress
RUN curl -o wordpress.tar.gz -SL \
  https://wordpress.org/wordpress-${WORDPRESS_VERSION}.tar.gz \
  && echo "$WORDPRESS_SHA1 *wordpress.tar.gz" | sha1sum -c - \
```

```
    && tar -xzf wordpress.tar.gz -C /usr/src/ \
    && rm wordpress.tar.gz \
    && chown -R www-data:www-data /usr/src/wordpress
COPY docker-entrypoint.sh /entrypoint.sh
# grr, ENTRYPOINT resets CMD now
ENTRYPOINT ["/entrypoint.sh"]
CMD ["apache2-foreground"]
```

下面是有关该 Dockerfile 的一些信息，你也许会感兴趣。

- 因为 WordPress 是基于 PHP 的，所以从 FROM 这一行可以识别出基础镜像是基于 Ubuntu 的 php 镜像（版本是 5.6-apache）。
- 第一个 RUN 指令加载了 Apache 的 rewrite 模块。
- 接下来的 RUN 指令运行了几个命令。它首先更新了容器中的所有 Ubuntu 软件包并安装了一些软件包。之后，它添加了 PHP 的 gd 扩展并设置了 PHP 使用的几个目录的位置。
- 接下来的 RUN 指令中的 `docker-php-ext-install` 命令安装了 PHP 的 mysqli 扩展。
- 通过 VOLUME 指令，`/var/www/html` 目录从宿主机挂载到容器中，用来存储 WordPress 使用的数据。
- 两条 ENV 指令设置了 WordPress 的版本号以及之后要使用的 SHA1SUM，在接下来的步骤中，当下载 WordPress 的 tarball 文件时，会使用该 SHA1SUM 检查下载的 tarball 文件是否有效。
- 接下来的 RUN 指令为获取和准备 WordPress 软件做了几件事情。`curl` 命令获取 wordpress-4.3.tar.gz 的 tarball 文件（使用前面设置的版本号），使用 SHA1SUM（也是之前设置的）对其进行检查，并将其解压到`/usr/src` 目录。接下来，它删除了该 tarball 文件并将`/usr/src` 目录的所有权修改为 `www-data` 用户和组。
- COPY 行将 `docker-entrypoint.sh` 脚本从构建目录复制为容器中的 `/entrypoint. sh` 文件。
- 最后，ENTRYPOINT 指令将`/entrypoint.sh` 设置为容器运行时启动的命令，而 CMD 将 `apache2-foreground` 设置为传递给 `entrypoint.sh` 脚本的选项。

要使用生成的镜像，要将 wordpress 容器与已经在系统中运行的 mysql 容器链接起来。可以按如下方式实现这一点：

```
# docker run --name new-wordpress --link new-mysql:mysql -d wordpress
```

wordpress 容器自身有一组环境变量，可以修改这组环境变量来改变 wordpress 容器的行为。可以通过`-e` 选项来设置 `docker run` 命令行的选项。下面是几个例子：

- -e WORDPRESS_DB_HOST：设置 MySQL 服务器的 IP 地址和端口。默认情况下，wordpress 会从链接的容器中找出 MySQL 服务。
- -e WORDPRESS_DB_USER=：修改用于访问 MySQL 数据库的用户名，默认用户

是 root。

- -e WORDPRESS_DB_PASSWORD=：修改 MySQL 数据库的 root 用户的密码。默认情况下使用 MYSQL_ROOT_PASSWORD 环境变量，该环境变量来自所链接的 MySQL 容器。
- -e WORDPRESS_DB_NAME=：修改 MySQL 数据库的名字，默认是 wordpress。

设置环境变量是许多容器镜像使用的技巧，以便你可以修改组件的位置、变更用户名和密码以及修改容器来适应你的环境。

18.2.2 查看 MySQL 的 Dockerfile

MySQL 是 Web 应用领域最流行的开源数据库。可以从 Docker Hub 获取用于构建 MySQL 官方镜像（mysql）的 Dockerfile（https://hub.docker.com/_/mysql/）。下面是最新版 mysql 镜像的 Dockerfile：

```
FROM debian:wheezy

RUN groupadd -r mysql && useradd -r -g mysql mysql
RUN mkdir /docker-entrypoint-initdb.d
RUN apt-get update && apt-get install -y perl --no-install-recommends && rm
➥ -rf /var/lib/apt/lists/*
RUN apt-key adv --keyserver ha.pool.sks-keyservers.net --recv-keys
A4A9406876FCBD3C456770C88C718D3B5072E1F5

ENV MYSQL_MAJOR 5.6
ENV MYSQL_VERSION 5.6.26

RUN echo "deb http://repo.mysql.com/apt/debian/ wheezy mysql-${MYSQL_MAJOR}"
➥  > /etc/apt/sources.list.d/mysql.list
RUN { \
echo mysql-community-server mysql-community-server/data-dir select ''; \
echo mysql-community-server mysql-community-server/root-pass password ''; \
echo mysql-community-server   mysql-community-server/re-root-pass password ''; \
echo mysql-community-server   mysql-community-server/remove-test-db select false; \
} | debconf-set-selections \
&& apt-get update && apt-get install -y mysql-server="${MYSQL_VERSION}"* &&
➥rm -rf /var/lib/apt/lists/* \
&& rm -rf /var/lib/mysql && mkdir -p /var/lib/mysql

RUN sed -Ei 's/^(bind-address|log)/#&/' /etc/mysql/my.cnf \
&& echo 'skip-host-cache\nskip-name-resolve' | awk '{ print } $1 ==
➥ "[mysqld]" && c == 0 { c = 1; system("cat") }' /etc/mysql/my.cnf > /tmp/
➥  my.cnf \
&& mv /tmp/my.cnf /etc/mysql/my.cnf

VOLUME /var/lib/mysql
COPY docker-entrypoint.sh /entrypoint.sh
```

```
ENTRYPOINT ["/entrypoint.sh"]
EXPOSE 3306
CMD ["mysqld"]
```

mysql 容器镜像的 Dockerfile 采用了许多 wordpress 镜像的 Dockerfile 中使用的技巧。下面描述了该文件的一些要点：

- FROM 行将 debian 镜像（版本是 wheezy）设置为基础镜像。
- 第一个 RUN 指令在容器内创建了 mysql 用户和用户组，而接下来的 RUN 指令创建了/docker-entrypoint-initdb.d 目录。
- 在更新了容器中的软件（apt-get update）后，接下来的 RUN 指令安装了 perl 软件包并清除了/var/lib/apt/lists/*目录。
- 在接下来的 RUN 指令中，apt-key 命令设置了用来验证 MySQL 软件有效性的键。
- 环境变量被设置（ENV 指令）来标识 MySQL 的主版本（MYSQL_MAJOR 5.6）和具体版本（MYSQL_VERSION 5.6.26）。
- 接下来的一行设置了 mysql 软件的位置（软件的名字中包含了前面设置的 MySQL 主版本号）。
- 接下来的 RUN 指令做了一些 MySQL 数据库的清理工作。
- 接下来的 RUN 指令做了更多的清理工作，修改了一些配置，并告诉 MySQL 不要对主机名进行反向查找。
- VOLUME 指令将 mysql 数据库的位置设置到宿主机的/var/lib/mysql 目录（这个指令会将该目录挂载到容器中的同一位置）。
- COPY 指令将构建目录中的 docker-entrypoint.sh 脚本复制为容器根目录中的/entrypoint.sh。
- MySQL 服务默认会监听 3306 端口，EXPOSE 指令将这个端口暴露给容器外部。
- 最后，CMD 指令将 mysqld 命令设置为默认运行的命令（这是 MySQL 的守护进程）。

与 wordpress 镜像一样，当运行 mysql 镜像时，你可以通过修改不同的环境变量来调整 MySQL 的使用方式以适应自己的情况。例如，可以设置下面这些选项。

- -e MYSQL_ROOT_PASSWORD=：设置用来访问 MySQL 数据库的 root 密码。
- -e MYSQL_DATABASE=：设置容器首次启动时要创建的 MySQL 数据库的名字。
- -e MYSQL_USER=：设置获得了数据库（用 MYSQL_DATABASE 变量创建）超级用户权限的用户的名字。
- -e MYSQL_PASSWORD=：确定 MYSQL_USER 所标识的用户账号的密码。
- MYSQL_ALLOW_EMPTY_PASSWORD：如果使用了这个可选变量（MYSQL_ALLOW_EMPTY_PASSWORD=yes），容器会使用空的 MySQL 密码启动。

如果要运行的容器镜像是由这里描述的 MySQL Dockerfile 构建的，可以通过向 docker run 命令行添加选项来调整容器的行为。除了设置刚刚描述的环境变量，还可以识别不同的

宿主机目录，用于提供 MySQL 数据库。例如：

```
# docker run --name new-mysql                \
   -v /my/custom:/etc/mysql/conf.d           \
   -v /my/own/datadir:/var/lib/mysql         \
   -e MYSQL_ROOT_PASSWORD=my-secret-pw       \
   -d mysql:latest
```

刚刚展示的 `docker run` 命令使用 `new-mysql` 容器名来运行 `mysql` 镜像。第一个卷挂载选项（`-v`）说明你要怎样使用宿主机上（`/my/custom` 目录）自己的 MySQL 配置文件，以及如何将它们挂载为容器中的配置文件（`/etc/mysql/conf.d` 目录）。

类似地，可以将宿主机上的数据文件（`/my/own/datadir` 目录）挂载到容器内的 MySQL 数据目录。这里的`-e` 选项将 MySQL 的 root 密码设置为 `my-secret-pw`。该命令行中最后的选项指明要运行 `mysql:latest` 镜像，并在后台以守护进程的方式（`-d`）运行该镜像。

18.3 细察桌面和个人使用的 Dockerfile

许多人是通过对桌面应用程序进行容器化开始了解如何使用 Docker 的。将桌面应用程序制作成容器有诸多优点。

如果想在同一个机器上使用一个应用程序的不同版本，将它们放入容器可以避免宿主机系统上潜在的库冲突。同样地，如果你之后决定不再在系统中保留该应用程序，只要停止容器并移除镜像，而不用搞乱你的系统。

在本节中，我会展示两种不同的方法——人们通过这些方法创建镜像，将容器化的 Web 浏览器添加到系统中。第一个例子展示了 Chrome 浏览器镜像，该镜像直接与宿主机的 X Window 系统的显示器进行集成。在第二个例子中，该镜像说明了如何使用 `vnc` 打开 Firefox 的虚拟桌面，或打开其他你想使用的 X Window 系统应用程序的虚拟桌面。

18.3.1 查看 Chrome 的 Dockerfile

谷歌公司的 Chrome 浏览器是 Linux 桌面的一个流行的附加应用，特别是那些只提供 Firefox 的 Linux 系统。Jessie Frazelle（https://blog.jessfraz.com），Docker 公司的一位雇员，创建了 Chrome 的容器镜像，可以从 Docker Hub 拉取该镜像，或者使用这里展示的 Dockerfile 自己构建该镜像。

这个 Chrome 镜像的工作方式是将桌面系统的 X11 套接字直接挂载到容器中。这样做可以避免使用 ssh 转发 X11 时带来的开销（既有规模上的，也有性能上的）。你还可以挂载音频设备（Debian 和 Ubuntu 中的`/dev/snd`），以便能通过浏览器直接播放音频。

下面是用来创建容器的 Dockerfile（从 https://github.com/jfrazelle/dockerfiles/blob/master/chrome/stable/Dockerfile 检查更新）：

```
FROM debian:sid
MAINTAINER Jessica Frazelle <jess@docker.com>

ADD https://dl.google.com/linux/direct/google-talkplugin_current_amd64.
➥ deb /src/google-talkplugin_current_amd64.deb

ADD https://dl.google.com/linux/direct/google-chrome-stable_current_
➥ amd64.deb /src/google-chrome-stable_current_amd64.deb

# Install Chromium
RUN mkdir -p /usr/share/icons/hicolor && \
        apt-get update && apt-get install -y \
        ca-certificates \
        gconf-service \
        hicolor-icon-theme \
        libappindicator1 \
        libasound2 \
        libcanberra-gtk-module \
        libcurl3 \
        libexif-dev \
        libgconf-2-4 \
        libgl1-mesa-dri \
        libgl1-mesa-glx \
        libnspr4 \
        libnss3 \
        libpango1.0-0 \
        libv4l-0 \
        libxss1 \
        libxtst6 \
        wget \
        xdg-utils \
        --no-install-recommends && \
        dpkg -i '/src/google-chrome-stable_current_amd64.deb' && \
        dpkg -i '/src/google-talkplugin_current_amd64.deb' \
        && rm -rf /var/lib/apt/lists/*

COPY local.conf /etc/fonts/local.conf

# Autorun chrome
ENTRYPOINT [ "/usr/bin/google-chrome" ]
CMD [ "--user-data-dir=/data" ]
```

下面是可以从 Chrome 的 Dockerfile 了解的信息。

- FROM 行标识出的基础镜像是 debian 镜像（版本是 sid）。
- 两条 ADD 指令从谷歌公司的下载网站复制了 Debian 软件包。通过将 google-talkplugin_current_amd64.deb 和 google-chrome- stable_current_amd64.deb 软件包复制到容器内的/src 目录，将这两个软件包添加进来。这两个软件包获取了 Google Talk 插件和最新稳定版的 Chrome 浏览器。

- 在创建图标目录并更新容器内的所有软件之后，RUN 指令安装了一堆软件包，这些软件包提供了容器内的声音和视频驱动。它还安装了从谷歌下载的那两个软件包。
- COPY 指令将构建文件夹的 `local.conf` 文件复制为容器内的`/etc/fonts/local.conf` 文件。
- 最后两条指令设置了要运行的命令。ENTRYPOINT 指令运行了 `google-chrome` 命令，以此启动浏览器，而 CMD 指令确定`/data` 目录作为浏览器获取用户数据的地方。

如果想构建这个镜像，可以从 GitHub 获取 Dockerfile 中定义的 `local.conf` 文件，该文件与 Dockerfile 文件放在相同的位置（https://github.com/jfrazelle/dockerfiles/tree/master/chrome/stable）。

谨记，这个 Dockerfile 为了在 Debian 风格的系统上运行而进行了裁剪。要使其能够用于不同的 Linux 系统，你可能需要修改宿主机上设备的位置并处理一些权限问题。（添加 `-privileged` 可以解决 Fedora 上的权限问题）。要完全避免这个问题，可以使用 VNC 或 X11 转发（如接下来的 Firefox 容器示例所展示的那样），但那个方案开销会更大。

下面是 Jess Frazelle 建议的运行 Chrome 容器镜像的 `docker run` 命令：

```
# docker run -it      \
   --net host         \
   --cpuset-cpus 0    \
   --memory 512mb     \
   -v /tmp/.X11-unix:/tmp/.X11-unix      \
   -e DISPLAY=unix$DISPLAY               \
   -v $HOME/Downloads:/root/Downloads    \
   -v $HOME/.config/google-chrome/:/data \
   --device /dev/snd \
   --name chrome      \
   jess/chrome
```

在这个例子中，有一些与容器内运行 Chrome 浏览器相关的有趣想法。`--cpuset-cpus` 和`--memory` 选项为限制 CPU 和内存的使用提供了一种简单的方法，由此可以减少 Chrome 对桌面性能所能造成的危害。

包含 X 套接字的目录（`/tmp/.X11-unix`）被挂载到容器内部，以提供对显示器的访问。DISPLAY 环境变量标识了本地显示器。方便起见，`/root/Downloads` 目录被挂载到容器中，这样，你下载的任何东西默认情况下都会保存到宿主机上。

Chrome 的设置和数据会保存到你的用户主目录中的`.config/google-chrome` 目录中，以便这些信息被持久保存。要让 Chrome 能够播放音频，`/dev/snd` 设备被挂载到容器中。图 18-1 展示了本地桌面上的容器所运行的 Chrome 浏览器。

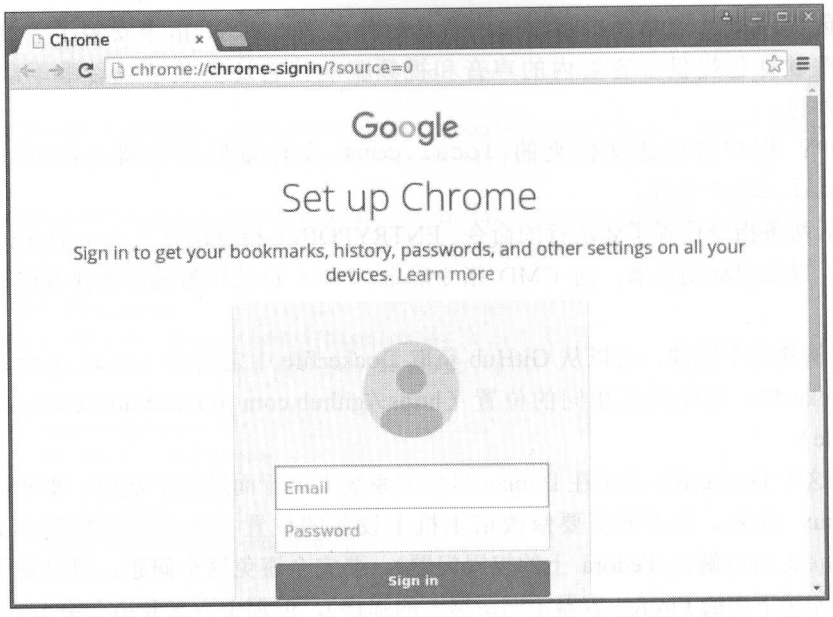


图 18-1 通过将宿主机的资源挂载到容器中的方式来从容器运行 Chrome

18.3.2 查看 Firefox 的 Dockerfile

Firefox 是 Linux 系统中的另一个流行的 Web 浏览器。Firefox 由 Mozilla 项目维护，其根源可以追溯到最初的 Web 浏览器。这里描述的 Firefox Dockerfile 是 fedora-cloud 倡议（https://github.com/fedora-cloud/Fedora-Dockerfiles）的一部分，该倡议创建了许多基于 Fedora 的容器镜像，这些镜像提供了独立的应用程序或其他容器用以构建的基础技术（如 Java 或 Python）。

Firefox 的 Dockerfile 的有趣之处在于，它将 Firefox 连同远程桌面软件（vnc）一起安装到容器镜像中。如同之前提到的，这样做的开销比直接将桌面套接字和设备挂载到容器中要大。但这种方法提供了一种更为通用的解决方案，其可以用于许多不同的桌面应用。

如果你想自己构建镜像，Firefox Dockerfile（https://github.com/fedora-cloud/Fedora- Dockerfiles/tree/master/firefox）需要在构建目录里添加一个 `xstartup` 脚本。该脚本与 Dockerfile 在 GitHub 的同一目录下。下面的例子展示了 Firefox 的 Dockerfile 到底是什么样子：

```
FROM fedora
MAINTAINER http://fedoraproject.org/wiki/Cloud

# Install the appropriate software
RUN yum -y update && yum clean all
RUN yum -y install firefox \
xorg-x11-twm tigervnc-server \
xterm xorg-x11-font \
```

```
xulrunner-26.0-2.fc20.x86_64 \
dejavu-sans-fonts \
dejavu-serif-fonts \
xdotool && yum clean all

# Add the xstartup file into the image and set the default password.
RUN mkdir /root/.vnc
ADD ./xstartup /root/.vnc/
RUN chmod -v +x /root/.vnc/xstartup
RUN echo 123456 | vncpasswd -f > /root/.vnc/passwd
RUN chmod -v 600 /root/.vnc/passwd

RUN sed -i '/\/etc\/X11\/xinit\/xinitrc-common/a [ -x /usr/bin/firefox ]
&& /usr/bin/firefox &' /etc/X11/xinit/xinitrc

EXPOSE 5901

CMD    ["vncserver", "-fg" ]
# ENTRYPOINT ["vncserver", "-fg" ]
```

下面解释了 Firefox 的 Dockerfile 做了什么。

- FROM 指示该容器起始于 fedora 基础镜像。
- 第一个 RUN 指令更新了容器中所有的 Fedora 软件（yum -y update）并清理了遗留的缓存文件。接下来的 RUN 指令安装了容器所需的软件包，包括浏览器本身（firefox）、一个小型窗口管理器（xorg-x11-twm）、远程桌面 VNC 服务器（tigervnc-server）、启动 Firefox 的软件（xulrunner）以及一些字体软件包。
- 为了存放 xstartup 脚本，接下来的 RUN 指令创建了/root/.vnc 目录。之后，ADD 指令将 xstartup 脚本从构建目录复制到容器中的/root/.vnc 目录。为了使该脚本能够执行，紧接着的 RUN 指令对该脚本运行了 chmod 命令。
- 接下来的 RUN 指令设置了 VNC 的密码。之后，当用 VNC 客户端尝试连接该容器时，需要提供 123456 这个密码。密码文件（/root/.vnc/passwd）被分配了由 root 用户读写的权限，而其他人没有被分配权限（chmod -v 600）。
- 紧接着的 RUN 指令通过 sed 命令将/usr/bin/firefox 命令添加到/etc/X11/xinit/xinitrc 文件中。
- EXPOSE 5901 指令对外开放了容器的 TCP 5901 端口，这为外部世界提供了访问容器内 VNC 服务的途径。
- 最后的指令（CMD）用来启动 vncserver 进程。

在继续之前，看一看 xstartup 脚本的内容是很有帮助的，VNC 服务器从容器启动时会执行这个脚本。下面是该脚本的最后几行：

```
if [ -f /etc/X11/xinit/xinitrc ]; then
  exec sh /etc/X11/xinit/xinitrc
```

```
fi
[ -r $HOME/.Xresources ] && xrdb $HOME/.Xresources
xsetroot -solid grey
xterm -geometry 80x24+10+10 -ls -title "$VNCDESKTOP Desktop" &
twm &
```

如果 xinitrc 文件存在（它确实存在），就执行该文件。Docker 之前将 firefox 命令添加到这个地方，因此要从这里运行它。如果$HOME/.Xresources 文件中保存了 X 的设置，那么 X 资源数据库实用工具（xrdb）运行的时候会包含这些设置。这之后，xsetroot 命令将 VNC 的背景色设置为灰色，一个有一定大小、带标题的 xterm（终端窗口）会被打开，而且其中会运行 twm 窗口管理器。如果想要改变 VNC 会话的外观或行为，你可以往这里添加命令或提供你自己的.Xresources 文件。

如果打算尝试一下 VNC 容器中的 Firefox 浏览器，从 GitHub 上获取 Dockerfile 和 xstartup 脚本，在存放这两个文件的目录中，运行下面的 docker build 命令：

```
# docker build -rm -t myfirefox .
...
Successfully built c5c9e7e770db
```

接着输入下面的命令，从容器运行 VNC 服务器和 Firefox：

```
# docker run -d -p 5901:5901 myfirefox
```

现在，vncserver 守护进程应该正在后台运行。

要尝试从你的本地系统运行 VNC 客户端。例如，你可以从桌面选择 Remote Desktop Viewer（远程桌面查看器），选择 Connect（连接），选择 VNC，在 Host（主机）输入框中输入 localhost:1，然后点击 Connect。当提示输入密码时，输入 123456。运行着 Firefox 的简单 VNC 桌面和 twm 窗口管理器就会出现。

要从命令行启动 VNC 客户端的话，可以从本地系统输入以下命令：

```
# yum install tigervnc -y
# vncviewer localhost:1
```

当提示时再次输入口令 123456，运行着 Firefox 的 VNC 桌面会出现在你的本地桌面上。图 18-2 展示的是一个运行在 Fedora 系统上的 VNC/Firefox 容器。

因为你是在 VNC 窗口中运行 Firefox，并没有将它当作容器化应用来运行，所以你保存的所有设置和文件都在容器中。如果要使用来自桌面系统的设置、将文件保存到桌面系统或连接到音响系统，你要挂载所需的卷、套接字和设备，就像在 Chrome 示例中所做的那样。

使用这里描述的 Dockerfile，可以修改它并用它启动你的 Linux 发行版中的任何桌面应用。

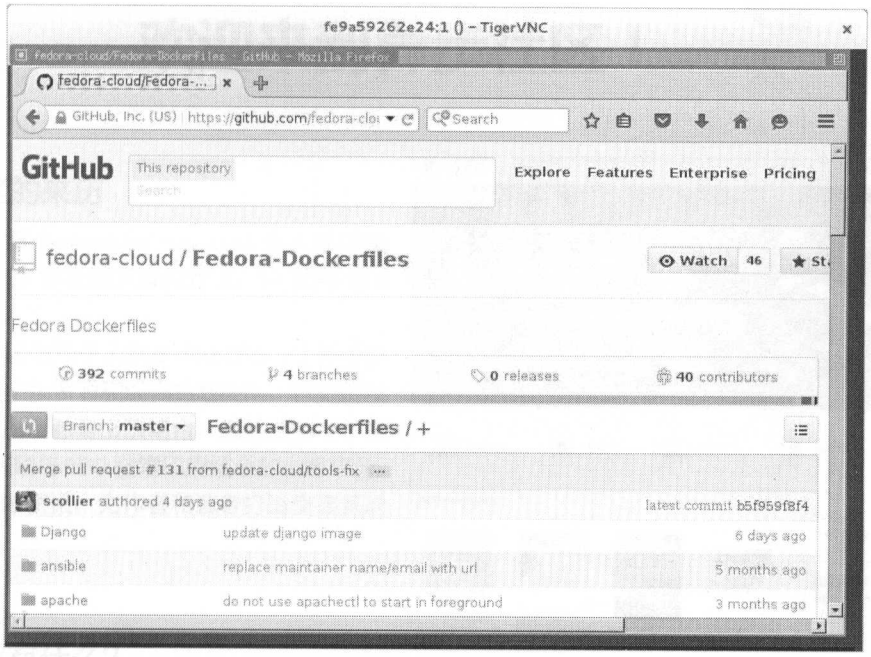


图 18-2 使用 VNC 容器来启动 Firefox 这样的桌面应用

18.4 小结

看其他人如何用 Dockerfile 创建镜像可以帮你弄清如何创建自己的 Docker 镜像。在本章中，我展示了几种如今正在使用的 Dockerfile，这些 Dockerfile 被用来创建基础镜像或创建用来运行特定应用程序的镜像。

就基础镜像而言，我展示了一个用来创建 CentOS 基础镜像的简单 Dockerfile。创建基础镜像的大多数工作是由 Kickstart 文件（最初安装所需的软件包）和 tar 命令（用来收集要放入容器的文件）完成的。一个更小的 Docker 基础镜像的例子是 busybox 镜像。与 CentOS 镜像一样，busybox 的 Dockerfile 多半只是拉入一个 tarball 文件（尽管只是一个大约 2.4 MB 的小文件）。

WordPress 和 MySQL 这样的开源应用项目在 Docker Hub 上提供了其官方 Docker 镜像。通过它们用于创建镜像的 Dockerfile 和其他文件，你可以看到环境变量的有趣用法——通过环境变量能够改变容器在运行的时候的使用方法。

本章最后描述的几个 Dockerfile 展示了通过容器在本地桌面上启动 Web 浏览器的不同方法。在通过容器使用 Chrome 浏览器的例子中，藉由挂载套接字、目录和音频设备将容器直接连接到桌面特性上；而容器中的 Firefox 浏览器这个例子却依赖 VNC 服务器启动一个精简版本的远程桌面，在其中管理 Firefox。

Docker容器 利用Kubernetes、Flannel、Cockpit和Atomic构建和部署

Linux系统或云环境上运行Docker的实用指南！

无论是在笔记本上还是在远程云上，Docker 都能够改变创建、测试、部署和管理最关键应用的方式。本书中，作者 Christopher Negus 帮助读者从头开始掌握 Docker 容器化技术。

开始的时候读者能够运行一些 Ubuntu、Fedora、RHEL、CoreOS 或 Project Atomic 的 Docker 容器镜像，看完本书之后，读者就可以在现代 Linux 和云环境中部署企业级质量、多容器的 Kubernetes。

本书是写给系统管理员、软件开发人员和技术狂热者的，书中触及了 Docker 使用的方方面面：设置容器化应用，使用单个和多个容器，在云环境中运行容器，以及开发容器。

书中通过桌面应用、系统服务和游戏等实际例子进行教学，指导读者构建和部署自己的 Docker 化应用。随着经验的充实，读者还会学到不可或缺的 Docker 最佳实践，这些实践用于构建和集成容器、进行日常的 Docker 管理，以及以下几个方面：

- 理解Docker是什么以及能用它来做什么；
- 在标准Linux上安装Docker，或者在Atomic Host和CoreOS这样的容器专用操作系统上安装Docker；
- 搭建容器运行环境和私有Docker registry；
- 创建、运行、探究Docker镜像和容器；
- 查找、拉取、保存、加载和标记容器镜像；
- 在本地系统和Docker registry之间拉取和推送容器；
- 将Docker容器与宿主机的网络和存储集成起来；
- 使用docker build命令和Dockerfile文件构建容器；
- 减少空间消耗并删除不需要的容器；
- 从容器内访问特定的宿主机权限；
- 通过Kubernetes将多个容器编排为复杂的应用；
- 在云环境中使用超级特权容器；
- 使用Cockpit管理云上的容器；
- 开始Docker容器开发；
- 通过共享的Dockerfile学习容器构建技巧。

Christopher Negus写了十多本关于Linux的书，包括畅销书《Red Hat Linux Bible》和《Linux Bible》。他是Red Hat认证架构师（RHCA）以及RHCE认证教练和考官。作为Red Hat的员工，他获得了RHCE和更高认证方面的Linux专业训练。目前，他是多个项目的主要作者，这些项目涉及Docker、Project Atomic、Kubernetes和OpenStack。

Pearson
www.pearson.com

异步社区
人民邮电出版社
www.epubit.com.cn

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

美术编辑：董志桢

分类建议：计算机／软件开发
人民邮电出版社网址：www.ptpress.com.cn

Pearson
ALWAYS LEARNING ALWAYS LEARNING ALWAYS LEARNING

ISBN 978-7-115-45105-7

ISBN 978-7-115-45105-7
定价：59.00 元